



Sonic[®]



Aurea[®] SonicMQ[®]
Application Programming Guide

Notices

For details, see the following topics:

- [Notices](#)
- [Third-party Acknowledgments](#)

Notices

Copyright © 1999 – 2015. Aurea Software, Inc. (“Aurea”). All Rights Reserved. These materials and all Aurea products are copyrighted and all rights are reserved by Aurea.

This document is proprietary and confidential to Aurea and is available only under a valid non-disclosure agreement. No part of this document may be disclosed in any manner to a third party without the prior written consent of Aurea. The information in these materials is for informational purposes only and Aurea assumes no responsibility for any errors that may appear therein. Aurea reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of Aurea to notify any person of such revisions or changes.

You are hereby placed on notice that the software, its related technology and services may be covered by one or more United States (“US”) and non-US patents. A listing that associates patented and patent-pending products included in the software, software updates, their related technology and services with one or more patent numbers is available for you and the general public’s access at www.aurea.com/legal/ (the “Patent Notice”) without charge. The association of products-to-patent numbers at the Patent Notice may not be an exclusive listing of associations, and other unlisted patents or pending patents may also be associated with the products. Likewise, the patents or pending patents may also be associated with unlisted products. You agree to regularly review the products-to-patent number(s) association at the Patent Notice to check for updates.

Aurea, Aurea Software, Actional, DataXtend, Dynamic Routing Architecture, Savvion, Savvion Business Manager, Sonic, Sonic ESB, and SonicMQ are registered trademarks of Aurea Software, Inc., in the U.S. and/or other countries. DataXtend Semantic Integrator, Savvion BizLogic, Savvion BizPulse, Savvion BizRules, Savvion BizSolo, Savvion BPM Portal, Savvion BPM Studio, Savvion Business Expert, Savvion ProcessEdge, and Sonic Workbench are trademarks or service marks of Aurea Software, Inc., in the U.S. and other countries. Additional Aurea trademarks or registered trademarks are available at: www.aurea.com/legal/.

The following third party trademarks may appear in one or more Aurea® Sonic® user guides:

Amazon is a registered trademark of Amazon Technologies, Inc.

Eclipse is a registered trademark of the Eclipse Foundation, Inc.

HP-UX is a registered trademark of Hewlett-Packard Development Company, L.P.

IBM, AIX, DB2, Informix, and WebSphere are registered trademarks of International Business Machines Corporation.

JBoss is a registered trademark of Red Hat, Inc. in the U.S. and other countries.

Linux is a registered trademark of Linus Torvalds.

Microsoft, Active Directory, Windows, and Visual Studio are registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Netscape is a registered trademark of AOL Inc.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

Progress and OpenEdge are registered trademarks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and other countries.

Red Hat and Red Hat Enterprise Linux are registered trademarks, and CentOS is a trademark of Red Hat, Inc. in the U.S. and other countries.

SUSE is a registered trademark of SUSE, LLC.

Sybase is a registered trademark of Sybase, Inc. in the United States and/or other countries.

Ubuntu is a registered trademark of Canonical Limited in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

All other marks contained herein are for informational purposes only and may be trademarks of their respective owners.

Third-party Acknowledgments

Please see the 'notices.txt' file for additional information on third-party components and copies of the applicable third-party licenses.

Table of Contents

Preface.....	15
About This Documentation.....	15
Typographical Conventions	17
Aurea® Sonic® Documentation.....	18
Aurea® SonicMQ® Documentation.....	18
Other Documentation in the Aurea® SonicMQ® Product Family.....	19
Worldwide Technical Support	20
 Chapter 1: Overview.....	 21
Java Message Service.....	22
JMS: Key Component of the Java Platform for the Enterprise.....	22
JMS Version 1.1 Specification.....	23
Java Development Environment.....	23
Programming Concepts.....	23
Clients Connect to the SonicMQ Broker.....	23
SonicMQ® Is a JMS Provider.....	24
Aurea® SonicMQ® Messaging Models.....	24
JMS Version 1.1 Unification of Messaging Models.....	25
SonicMQ® Object Model.....	26
Quality of Service and Protection.....	28
Clients.....	31
Java Client.....	31
JMS Test Client.....	31
HTTP Direct Protocol Handlers.....	31
Java Applet.....	32
.NET Client.....	32
C/C++ Clients.....	32
COM Client.....	32
SonicMQ API.....	32
 Chapter 2: Using the JMS Test Client.....	 33
Testing Point-to-point Messaging.....	33
Starting the SonicMQ® Container and Broker.....	34
Opening the JMS Test Client.....	34
Establishing Connection to the SonicMQ Broker.....	35
Establishing a Queue Session.....	35
Creating Queue Senders and Queue Receivers.....	36
Sending and Receiving Messages.....	38

Browsing Messages on a Queue.....	40
Testing Publish and Subscribe Messaging.....	42
Establishing a Topic Session.....	43
Creating Publishers and Subscribers to Topics.....	44
Publishing Messages.....	46
Receiving Messages on Subscribed Topics.....	49
 Chapter 3: Examining the SonicMQ® JMS Samples.....	51
About SonicMQ® Samples.....	52
Other Samples Available.....	54
Extending the Samples.....	55
Running the SonicMQ Samples.....	56
Starting the SonicMQ® Container and Management Console.....	56
Opening the Sonic Management Console from a Linux or UNIX console window.....	58
Using the Sample Scripts.....	58
Chat and Talk Samples.....	59
Chat Application (Pub/Sub).....	59
Talk Application (PTP).....	60
Reviewing the Chat and Talk Samples.....	60
MultiTopicChat Sample.....	61
Setting Up MultiTopic Sessions.....	61
Demonstrating MultiTopic Publish and Subscribe.....	61
Samples of Additional Message Types.....	62
Map Messages (PTP).....	63
XML Messages.....	64
Decomposing Multipart Messages.....	67
Reviewing the Additional Message Type Samples.....	68
Sample of Channels for Large Message Transfers.....	68
Reviewing the Large Message Transfer Sample.....	70
Message Traffic Monitor Samples.....	70
QueueMonitor Application (PTP).....	70
MessageMonitor Application (Pub/Sub).....	72
Transaction Samples.....	73
TransactedTalk Application (PTP).....	74
TransactedChat Application (Pub/Sub).....	75
Reviewing the Transaction Samples.....	76
Reliable, Persistent, and Durable Messaging Samples.....	77
Reliable Connections.....	77
Persistent Storage Application (PTP).....	79
DurableChat Application (Pub/Sub).....	86
Continuous Producer Demonstrating Client Persistence.....	88
Reviewing Reliable, Persistent, and Durable Messaging.....	91
Request and Reply Samples.....	91
Request and Reply (PTP).....	92

Request and Reply (Pub/Sub).....	92
Reviewing the Request and Reply Samples.....	93
Selection, Group, and Wild Card Samples.....	93
Message Selection: SelectorTalk and SelectorChat	93
MessageGroupTalk (PTP).....	95
HierarchicalChat Application (Pub/Sub).....	98
Running HierarchicalChat.....	99
Reviewing the Selection, Group, and Wild Card Samples.....	99
Test Loop Sample.....	100
QueueRoundTrip Application (PTP).....	100
Enhancing the Basic Samples.....	100
Use Common Topics Across Clients.....	100
Trying Different RoundTrip Settings.....	101
Modifying the MapMessage to Use Other Data Types.....	102
Modifying the XMLMessage Sample to Show More Data.....	103

Chapter 4: SonicMQ® Connections.....107

Overview of SonicMQ Connections.....	108
Protocols.....	110
TCP.....	110
SSL.....	110
HTTP.....	113
HTTPS	114
JVM Command Options.....	114
HTTP Tunneling through an Authenticating Proxy.....	114
HTTP Forward Proxy.....	115
HTTPS Forward Proxy.....	115
SSL/HTTPS.....	116
Nagle Algorithm.....	116
HTTP Map Host to IP.....	117
Connection Factories and Connections.....	117
Connection Factories.....	117
Connecting to SonicMQ Directly.....	124
Connecting to SonicMQ Using Administered Objects.....	124
Connecting to SonicMQ Using Serialized Factories.....	129
Connections.....	130
Client Persistence.....	133
Using Client Persistence.....	134
Rejection Listener.....	135
Coding Limitations.....	135
Asynchronous Message Delivery.....	136
Delivery Mode Behavior.....	136
Reliability of Produced Messages.....	137
Ordering of Asynchronously Produced Messages.....	138

Delivery Doubt Window.....	138
Close Behavior.....	139
RejectionListener Semantics.....	140
Fault-Tolerant Connections.....	140
How Fault-Tolerant Connections are Initially Established.....	141
ConnectionFactory Methods for Fault-Tolerance.....	142
Connection Methods for Fault-Tolerance.....	145
Reconnect Errors.....	148
Load Balancing Considerations.....	148
Forward and Reverse Proxies.....	149
Client Persistence and Fault-Tolerant Connections.....	149
JMS Operation Reliability and Fault-Tolerant Connections.....	151
Reconnect Conflict.....	151
Message Reliability.....	152
NON_PERSISTENT_REPLICATED Delivery Mode.....	153
Modifying the Chat Example for Fault-Tolerance.....	160
Starting, Stopping, and Closing Connections.....	163
Starting a Connection.....	163
Stopping a Connection.....	163
Closing a Connection.....	164
Using Multiple Connections.....	164
Communication Layer.....	165

Chapter 5: SonicMQ Client Sessions.....167

Overview of Client Sessions.....	167
Naming Sessions.....	168
Acknowledgement Mode.....	169
Explicit Acknowledgement.....	171
Transacted Sessions.....	171
Distributed Transactions.....	172
Duplicate Message Detection.....	172
Session Objects.....	173
Creating a Destination.....	174
Creating a MessageProducer.....	177
Creating a MessageConsumer.....	177
Creating a Message.....	178
Closing a Session.....	179
Flow Control.....	179
Using Client Persistence and Wait Time When Flow Controlled.....	180
Flow Control Management Notifications.....	180
Disabling Flow Control.....	182
Flow to Disk.....	182
Send Timeout.....	183
Using Sessions and Consumers.....	183

Multiple Sessions on a Connection.....	184
Creating Session Objects and the Listeners.....	184
Starting the Connection.....	185
JMS Messaging Domains.....	185
Integration with Application Servers.....	186
Connection Consumer.....	187
XA Resources.....	188
Chapter 6: Messages.....	191
About Messages.....	191
Message Type.....	192
Creating a Message.....	193
Working With Messages That Have Multiple Parts.....	197
Composition of a MultipartMessage.....	197
Using Multipart Messages to Wrap Problem Messages.....	203
Interacting with Business-to-Business Multipart Types	204
Message Structure.....	205
Message Header Fields.....	205
Message Properties.....	208
Provider-defined Properties (JMS_SonicMQ).....	208
JMS-defined Properties (JMSX).....	210
User-defined Properties.....	211
Setting Message Properties.....	211
Property Methods.....	212
Message Body.....	213
Setting the Message Body.....	214
Getting the Message Body.....	214
Chapter 7: Message Producers and Consumers.....	215
About Message Producers and Message Consumers.....	216
Message Ordering and Reliability.....	217
Destinations.....	218
Steps in Message Production.....	218
Create a Session.....	218
Create the Producer on the Session.....	219
Create the Message Type and Set Its Body.....	219
Set Message Header Fields.....	219
Set the Message Properties.....	220
Elect Per Message Encryption.....	220
Produce the Message.....	220
Message Management by the Broker.....	221
Message Receivers, Listeners, and Selectors.....	223
Message Receiver.....	223

Message Listeners.....	224
Message Selection.....	224
Steps in Listening, Receiving, and Consuming Messages.....	228
Implement the Message Listener	228
Create the Destination and Consumer, Then Listen.....	228
Handle a Received Message.....	228
Reply-to Mechanisms.....	230
Temporary Destinations Managed by a Requestor Helper Class.....	230
Producers and Consumers in JMS Messaging Domains.....	232

Chapter 8: Point-to-point Messaging.....235

About Point-to-point Messaging.....	236
Message Ordering and Reliability in PTP	237
Message Ordering.....	237
Message Delivery.....	238
Using Multiple MessageConsumers.....	238
Message Queue Listener.....	239
MessageConsumer.....	239
Using Message Grouping.....	240
Illustration of Message Grouping.....	240
Broker Settings for Message Grouping.....	241
Message Producers for Message Grouping.....	242
Message Consumers for Message Grouping.....	243
Setting Prefetch Count and Threshold.....	243
Browsing a Queue.....	244
Handling Undelivered Messages.....	246
Setting Important Messages to be Saved if They Expire	246
Setting Small Messages to Generate Administrative Notice.....	246
Life Cycle of a Guaranteed Message.....	247
Setting the Message to Be Preserved.....	247
Setting the Message to Generate an Administrative Event.....	247
Sending the Message.....	247
Letting the Message Get Delivered or Expire.....	248
Post-processing Expired Messages.....	248
Getting Messages Out of the Dead Message Queue.....	249
Detecting Duplicate Messages.....	249
Forwarding Messages Reliably.....	250
Modifying a sample to show the acknowledgeAndForward behavior.....	251
Dynamic Routing with PTP Messaging.....	251
Administrative Requirements	252
Application Programming Requirements.....	252
Message Delivery with Dynamic Routing.....	252
Clusterwide Access to Queues.....	253
Sending to Clusterwide Queues.....	253

Receiving from Clusterwide Queues.....	254
Browsing Clusterwide Queues.....	254
Message Selectors with Clusterwide Queues.....	254
Clustered Queue Availability When Broker is Unavailable.....	254

Chapter 9: Publish and Subscribe Messaging.....255

About Publish and Subscribe Messaging.....	256
Message Ordering and Reliability in Pub/Sub.....	257
General Services.....	258
Message Ordering.....	258
Reliability.....	258
Topic.....	259
MessageProducer (Publisher).....	259
Creating the MessageProducer.....	260
Creating the Message.....	260
Sending Messages to a Topic.....	260
MessageConsumer (Subscriber).....	260
Durable Subscriptions.....	261
Clusterwide Access to Durable Subscriptions.....	262
Dynamic Routing with Pub/Sub Messaging.....	264
Administrative Requirements	265
Application Programming Requirements.....	265
Message Delivery with Remote Publishing.....	266
Shared Subscriptions.....	266
Features of Using Shared Subscriptions in Your Applications.....	268
Usage Scenarios for Shared Subscriptions.....	269
Defining Shared Subscription Topic Subscribers	270
Message Delivery to a Broker with Shared Subscriptions.....	272
JMS Interactions with Shared Subscriptions.....	276
Shared Subscriptions with Remote Publishing and Subscribing	277
MultiTopics.....	280
Format of a MultiTopic String.....	280
Creating MultiTopics.....	281
Adding Component Topics to a MultiTopic.....	282
Publishing and Subscribing to MultiTopics.....	282
MultiTopic Considerations.....	284

Chapter 10: Guaranteeing Messages.....287

Introduction.....	287
Duplicate Message Detection Overview.....	288
SonicMQ Extensions to Prevent Duplicate Messages.....	288
Support for Detecting Duplicate Messages.....	289
Dead Message Queue Overview.....	289

What Is an Undeliverable Message?.....	290
Using the Dead Message Queue.....	290
Monitoring Dead Message Queues.....	291
Default DMQ Properties.....	291
JMS_SonicMQ Message Properties Used for DMQ.....	292
Setting the Message Property to Preserve If Undelivered.....	293
Handling Undelivered Messages.....	294
Sample Scenarios in Handling Dead Messages.....	294
What To Do When the Dead Message Queue Fills Up.....	295
Undelivered Messages Due to Expired TTL.....	295
Specifying a Destination for Undelivered Messages.....	296
How to Specify an Undelivered Destination.....	296
Failure to Forward Undelivered Messages to the Undelivered Destination.....	300
Publish Permission Check.....	300
Undelivered Message Notifications.....	301
Undelivered Destinations for DRA Messages.....	301
Undelivered Message Reason Codes.....	302

Chapter 11: Recoverable File Channels.....307

About Recoverable File Channels for Large Messages.....	307
Forwarding the Header Message.....	308
Global Queues.....	309
Classes and Interfaces for Large Message Transfers.....	310
General Procedure for Large Message Transfers.....	316
Creating a Recoverable File Channel.....	317
Recovering an Interrupted Transfer.....	318
Duplicate Detection for File Transfers.....	320
Security on File Transfers.....	321
Using Multiple File Channels.....	321
Exception Handling for File Channels.....	321
Log Files.....	323
Tips and Techniques for Using File Channels.....	323

Chapter 12: SonicStream API.....325

About the SonicStream API	325
Common SonicStreamFactory Semantics.....	326
Constructors.....	327
Methods.....	327
SonicStream Interface.....	327
Stream Publisher Semantics.....	328
SonicStreamFactory.....	328
SonicOutputStreamController Interface.....	329
Stream Subscriber Semantics.....	330

SonicStreamFactory.....	330
Managing Flow Control.....	332
Handling Errors.....	333
Samples of SonicStreams.....	333
SonicStreams Sample.....	334
SonicStreams Sample With Retry	335
 Chapter 13: Hierarchical Name Spaces.....	345
About Hierarchical Name Spaces.....	345
Advantages of Hierarchical Name Spaces.....	346
Publishing a Message to a Topic.....	347
Topic Notation that Enables Topic Hierarchies.....	347
Broker Management of Topic Hierarchies.....	348
Subscribing to Nodes in the Topic Hierarchy.....	348
Template Characters.....	349
Examples of a Topic Name Space.....	353
Publishing Messages to a Hierarchical Topic.....	353
Subscribing to Sets of Hierarchical Topics.....	353
 Chapter 14: Distributed Transactions Using XA Resources.....	355
About Distributed Transactions.....	355
General Properties of a Transaction.....	356
Transaction Types.....	356
Components of Distributed Transactions.....	356
Using XA Resources.....	357
Interfaces for Distributed Transactions.....	359
javax.transaction.xa Interfaces.....	359
JMS XA SPI Interface.....	359
In-doubt Global Transactions.....	361
SonicMQ Can Complete In-doubt Transaction Branches.....	361
Access Control Group for Transaction Administrators.....	361
Transaction Recovery.....	361
Distributed Transactions Models.....	363
SonicMQ Integrated with an Application Server.....	363
SonicMQ Directly Used with a Transaction Manager.....	365
SonicMQ Performing DTP Without a Transaction Manager.....	366
Running the Distributed Transaction Sample.....	368
Creating a distributed transaction.....	368
Creating receivers for a distributed transaction.....	369
Committing or rolling back the distributed transaction.....	369
Consuming messages from a distributed transaction.....	370
 Chapter 15: Using the Sonic JNDI SPI.....	371

Overview of the JNDI SPI.....	371
Sonic JNDI SPI Samples.....	374
Java JNDI SPI Sample.....	374
JavaScript JNDI API Samples.....	376
 Chapter 16: Using Client Tracing Logs.....	377
Overview of SonicMQ JMS API Tracing	377
Enabling JMS Tracing.....	378
Trace Levels.....	378
Setting the Trace Level in Applications.....	379
Exploring Tracing in the SonicMQ Sample Applications.....	379
Using Tracing in the Sample Application Chat	380
 Index.....	383

Preface

For details, see the following topics:

- [About This Documentation](#)
- [Typographical Conventions](#)
- [Aurea® Sonic® Documentation](#)
- [Aurea® SonicMQ® Documentation](#)
- [Worldwide Technical Support](#)

About This Documentation

This guide is part of the documentation set for Aurea® SonicMQ® 2015 SP1.

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

Aurea® SonicMQ® is a complete implementation of the Java Message Service specification Version 1.1, an API for accessing enterprise messaging systems from Java programs.

This book provides the information a Java software developer needs to use the application program interfaces to create Aurea® SonicMQ® client applications.

The sample software provided in source form on the SonicMQ media is the basis for the discussions of features and concepts.

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

- [Overview](#) discusses the environment and Java constructs that can be used in messaging applications. The basic concepts in this chapter set the groundwork for understanding how to

build efficient applications. The service and protection features in SonicMQ are presented in a tabular form with references to other chapters and other books for implementation details.

- [Using the JMS Test Client](#) describes how to use the JMS Test Client to examine both Publish and Subscribe and Point-to-point messaging.
- [Examining the SonicMQ JMS Samples](#) takes an in-depth tour through the console-based code samples introduced in the *Getting Started with Aurea SonicMQ* manual, focusing on the programming functions and features used.
- [SonicMQ Connections](#) explores protocols, connection factories, connections. The identifiers and parameters of connections are presented. The techniques for direct creation of factories are contrasted to the ways that administered objects can be used in serialized Java objects and LDAP lookup through JNDI on the built-in or external LDAP stores.
- [SonicMQ Client Sessions](#) on page 167 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.
- [Messages](#) on page 191 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. Also describes the XML message and Multipart message.
- [Message Producers and Consumers](#) describes the scope of the session objects that produce messages and the session objects that listen, receive, and consume messages.
- [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.
- [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 as well as advanced features such as remote publishing, shared subscriptions, and multi-topics are presented in depth.
- [Guaranteeing Messages](#) on page 287 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.
- [Recoverable File Channels](#) describes the Point-to-point feature that provides fully recoverable transfers of files between peers through SonicMQ brokers and common global queues.
- [SonicStream API](#) describes this API lets you send streams of data to interested applications, using SonicMQ as the transport mechanism.
- [Hierarchical Name Spaces](#) on page 345 explains SonicMQ's topic hierarchies and how they can be used to streamline access to data.
- [Distributed Transactions Using XA Resources](#) explains distributed transaction processing, and presents several distributed transaction models and describes how to run the distributed transaction sample.

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 **monospacetypeface** 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.

Aurea[®] Sonic[®] Documentation

Aurea[®] Sonic[®] platform installations always have a welcome page that provides links to Aurea[®] Sonic[®] documentation, release notes, communities, and support. See the *Product Update Bulletin* for "what's new" and "what's changed" since prior releases.

The Aurea[®] Sonic[®] documentation set includes the following books and API references.

Aurea[®] SonicMQ[®] Documentation

Aurea[®] SonicMQ[®] software installations provide the following documentation:

- *Aurea[®] Sonic[®] Installation and Upgrade Guide* — The essential guide for installing, upgrading, and updating Aurea[®] SonicMQ[®] software 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.
- *Aurea[®] SonicMQ[®] Getting Started Guide* — Provides an introduction to the scope and concepts of Aurea[®] SonicMQ[®] messaging. Describes the features and benefits of Aurea[®] 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 Aurea[®] SonicMQ[®] documentation set.
- *Aurea[®] SonicMQ[®] Configuration and Management Guide* — Describes the configuration toolset for objects in a domain. Also shows how to use the JNDI store for administered objects, how integration with Aurea[®] Actional[®] platform is implemented, and how to use JSR 160 compliant consoles. Shows how to manage and monitor deployed components including metrics and notifications.
- *Aurea[®] SonicMQ[®] Deployment Guide* — Describes how to architect components in broker clusters, the Aurea[®] 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 Aurea[®] Sonic[®] technique that enables Aurea[®] SonicMQ[®] brokers to send and receive pure HTTP messages.
- *Aurea[®] SonicMQ[®] Administrative Programming Guide* — Shows how to create applications that perform management, configuration, runtime and authentication functions.
- *Aurea[®] 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.
- *Aurea[®] 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 Aurea[®] Sonic[®] Continuous Availability Architecture[™].

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

Other Documentation in the Aurea® SonicMQ® Product Family

The Aurea® Sonic® download site provides access to additional client and JCA adapter products and documentation:

- *Aurea® SonicMQ® .NET Client Guide* — Packaged with the Aurea® 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 Aurea® Sonic® .NET client libraries, and samples for C++ and VB.NET.
- *Aurea® SonicMQ® C Client Guide* — Packaged with the Aurea® 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 Aurea® SonicMQ® C client.
- *Aurea® SonicMQ® C++ Client Guide* — Packaged with the Aurea® 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 Aurea® SonicMQ® C++ client.
- *Aurea® SonicMQ® COM Client Guide* — Packaged with the Aurea® 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 Aurea® SonicMQ® COM client.
- *Aurea® SonicMQ® Resource Adapter for JCA - User's Guide for WebSphere®* — Packaged with this JCA adapter in a separate download, this guide describes the Aurea® Sonic® Resource Adapter for JCA and using it with a WebSphere® application server.

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

Worldwide Technical Support

Aurea® Software's support staff can provide assistance from the resources on their web site at <http://www.aurea.com/sonic>. There you can access technical support for licensed Aurea® Sonic® products to help you resolve technical problems that you encounter when installing or using Aurea® Sonic® products.

When contacting Technical Support, please provide the following information:

- The release version number and serial number of Aurea® SonicMQ® that you are using. This information is listed on the license addendum. It is also at the top of the Aurea® SonicMQ® Broker console window and might appear as follows:

```
Aurea® SonicMQ® Continuous Availability Edition [Serial Number nnnnnnnn]  
Release nnn Build Number nnn Protocol nnn
```

- The release version number and serial number of Aurea® Sonic ESB® that you are using. This information is listed on the license addendum. It is also near the top of the console window for an Aurea® Sonic ESB® Container. For example:

```
Sonic ESB Continuous Availability Edition [Serial Number nnnnnnnn]  
Release nnn Build Number nnn Protocol nnn
```

- The platform on which you are running Aurea® Sonic® products, and any other relevant environment information.
- The Java Virtual Machine (JVM) your installation uses.
- Your name and, if applicable, your company name.
- E-mail address, telephone, and fax numbers for contacting you.

Overview

SonicMQ is Aurea Software's implementation of Sun's Java Message Service (JMS) specification that expedites development and deployment of 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
 - A JNDI store is provided for administered objects

- SonicMQ provides secure data transmission and controlled access with:
 - Pluggable cipher suites for Quality of Protection (QoP)
 - Pluggable client authentication
 - Option to use LDAP as the repository of user names and passwords
- Flexibility in configuring the messaging infrastructure:
 - Clients can be moved around the network without requiring any changes to the messaging application
 - Support for XML message types in addition to the JMS types
 - 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 learned and deployed

For details, see the following topics:

- [Java Message Service](#)
- [Programming Concepts](#)
- [Quality of Service and Protection](#)
- [Clients](#)
- [SonicMQ API](#)

Java Message Service

The Java Message Service (JMS) Version 1.0.2b specification describes portable, efficient standards for a powerful, extensible messaging service. The JMS specification pointedly leaves some functionality—such as load balancing, fault tolerance, error notification, administration, security, wire protocol, and message repository—to the provider of the messaging server. SonicMQ implements this functionality and provides a level of abstraction to developers, who can concentrate on creating business logic.

JMS: Key Component of the Java Platform for the Enterprise

Sun Microsystems announced a plan in early 1997 to deliver nine Java APIs that would enable a vendor-neutral computing infrastructure capable of integrating Java with virtually every significant enterprise computing service.

JMS would provide asynchronous communications to avoid the problems synchronous communications—such as RMI and CORBA—were experiencing in the uncontrollable Internet environment. Javasoft provided a reference implementation in late 1998, noting that implementers of the JMS specification would need to match the security, reliability, fault-tolerance, and manageability of existing mainframe messaging services before enterprise acceptance would be considered. At the 1.3 release of the Java 2 Enterprise

Edition platform, the JMS API is an integral part of the platform. JMS is a strategic technology for J2EE. JMS will work in concert with other technologies to provide reliable, asynchronous communication between components in a distributed computing environment. The JMS specification notes that it does not address load balancing, fault tolerance, error notification, administration, security, and repositories.

JMS Version 1.1 Specification

In April, 2002, Sun introduced Version 1.1 of the JMS specification. The main enhancement in this specification is the refactoring of interfaces to support “**domain unification**.” In JMS 1.02b, there was a strong distinction between the Point-to-Point and Pub/Sub messaging models (referred to as **messaging domains** in the JMS 1.1 specification), each requiring its own set of interfaces. One consequence of this separation was that a single transaction could not include both Point-to-Point and Pub/Sub messages. In JMS Version 1.1, this constraint has been removed, and it is now possible to include messages from both models in a single transaction.

The JMS 1.1 specification describes a common set of interfaces that can be used for both messaging models. Because of this, applications written to the JMS 1.1 API can safely ignore interfaces that were previously required. The reduced number of interfaces simplifies application code. It can also eliminate redundant code.

JMS Version 1.1 is **fully backwards compatible** with JMS Version 1.02b. Client code that conforms to the JMS 1.02b specification also conforms to the JMS 1.1 specification.

Java Development Environment

SonicMQ is delivered with a Java run-time environment (JRE) consisting of a Java Virtual Machine (JVM) that is sufficient to support the Java-based installer and the demonstration of SonicMQ samples running against an embedded persistent storage mechanism.

Important: The installable JVM might not be appropriate on every platform. See the *SonicMQ Release Notes* in the **docs** folder of your SonicMQ installation to get detailed information about the JVM that is appropriate for your platform, operating system, persistent storage mechanism, and toolset

Programming Concepts

The design of SonicMQ provides full implementation of the Java Message Service (JMS) specification with additional features that comprise a solution that is resilient enough for Internet E-commerce in major enterprises.

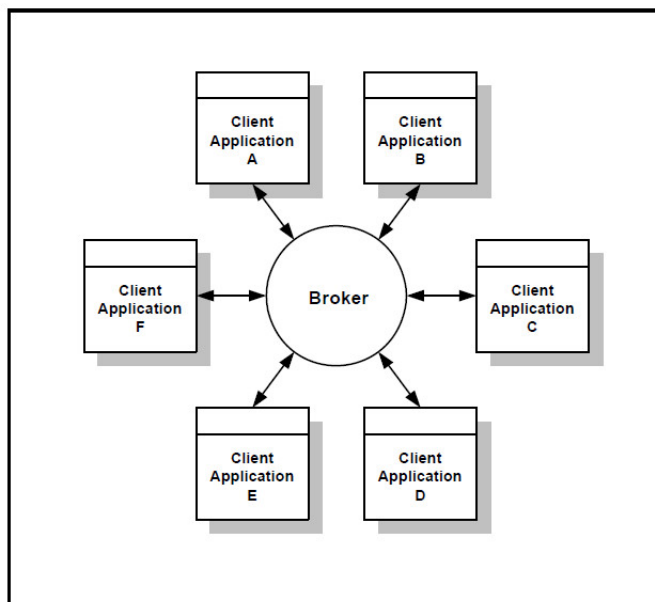
Messaging involves the loose coupling of applications. This is accomplished by maintaining an intelligent broker structure. A client can establish one or more connections to a broker.

Clients Connect to the SonicMQ Broker

In the following figure, 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.

The SonicMQ communication layer abstracts developers from the plumbing of the underlying network, freeing them to concentrate on constructing business logic in Java applications.

Figure 1: Broker Is a Hub for SonicMQ Client Applications

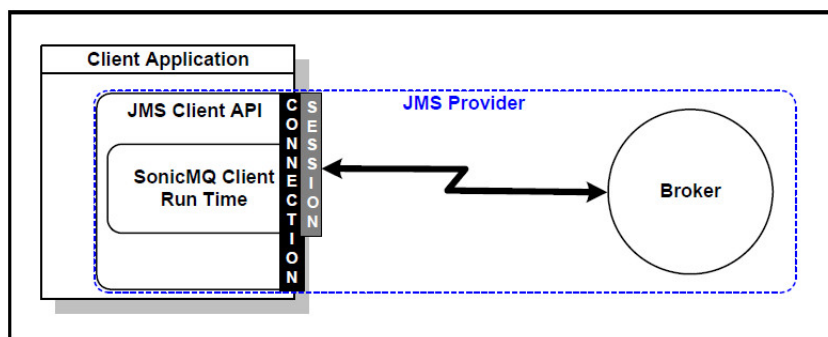


The broker can join with other brokers to form **clusters**. Clusters and stand-alone brokers are nearly equivalent when looked at as routing nodes.

SonicMQ[®] Is a JMS Provider

The components that are needed to implement and manage a JMS application are supplied by the **JMS provider**. This includes, as shown in the following figure, the JMS Client API and the SonicMQ[®] Client Run Time accessed from within the client application, the communications layer between the client and the broker architecture—repositories (message, security, and configuration), and administrative tools for managing clusters, security, administered objects, and the brokers.

Figure 2: Client Application Using the SonicMQ JMS Provider



Aurea[®] SonicMQ[®] Messaging Models

There are two **messaging models** in SonicMQ:

- **Point-to-point (PTP)** — In this model, the producer of a message sends a message to a specified static queue at a broker. While many prospective recipients could be listening to or even browsing the queue, when a receiver elects to accept a queued message, the message

is considered delivered. No other recipient will thereafter be able to access that message. PTP is a *one-to-one* form of communication.

- **Publish and Subscribe (Pub/Sub)** — In this model, the producer of a message sends the message to a specified topic at the broker. Pub/Sub is referred to as *one-to-many* or *broadcast* because there could be zero to many subscribers for a given topic who will each receive the one message that was sent.

JMS Version 1.1 Unification of Messaging Models

Prior to JMS Version 1.1, the Point-to-Point and Pub/Sub messaging models were kept separate, and each model required its own set of interfaces. Although the model-specific interfaces extended a common base set of interfaces, it was impossible to use the common interfaces to implement functionality that was specific to either model. The common interfaces and their model-specific extensions are shown in the following table:

Table 1: Common and Model-Specific Interfaces (Continued)

Common	Point-to-Point	Pub/Sub
Connection	QueueConnection	TopicConnection
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Destination	Queue	Topic
MessageConsumer	QueueReceiver	TopicSubscriber
MessageProducer	QueueSender	TopicPublisher
Session	QueueSession	TopicSession

In JMS Version 1.1, the common interfaces were enhanced, allowing application programmers to use the common interfaces to directly implement model-specific functionality, rather than using the model-specific interfaces. These enhancements provide two important benefits:

- A single transaction can now include Point-to-Point and Pub/Sub messages.
- The JMS programming model is simplified. Application programmers can now focus on the common interfaces, without being forced to use two model-specific interfaces. Also, if a single application requires both Point-to-Point and Pub/Sub functionality, the application programmer is no longer forced to create redundant code.

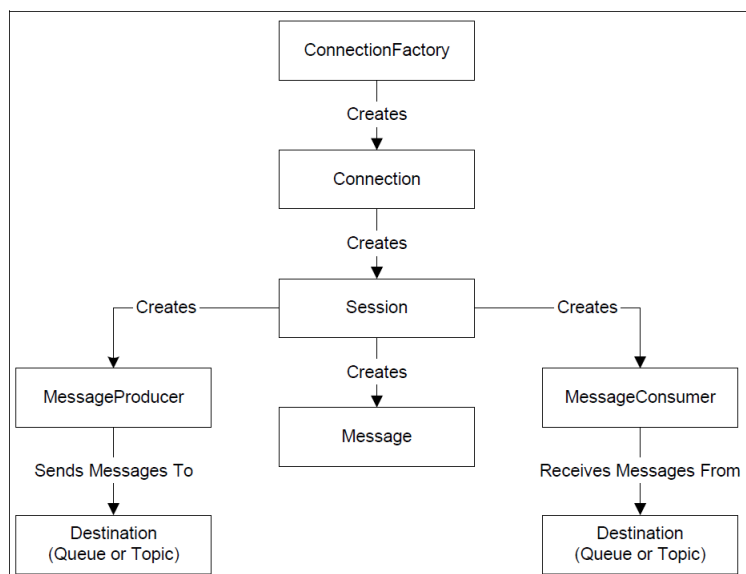
Although the JMS 1.1 common interfaces effectively replace many of the model-specific interfaces, the model-specific interfaces continue to be fully supported. This makes the JMS 1.1 API **fully backwards compatible**. Any JMS application written before JMS 1.1 will continue to work as expected.

Despite the fact that the model-specific interfaces continue to be supported, the JMS 1.1 specification states that some interfaces might be deprecated in the future. Consequently, if you are developing new JMS client applications, it is recommended that, wherever possible, you use the common interfaces in place of the older model-specific interfaces.

SonicMQ® Object Model

The following figure shows the SonicMQ object model.

Figure 3: SonicMQ Object Model



ConnectionFactory

A **ConnectionFactory** is an object whose job is to create one or more **Connection** objects, each of which establishes a connection to a SonicMQ broker (or cluster). A **ConnectionFactory** can be implemented as an administered object.

Connection

A **Connection** is a conduit for communication between your client application and a SonicMQ broker (or cluster). 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.

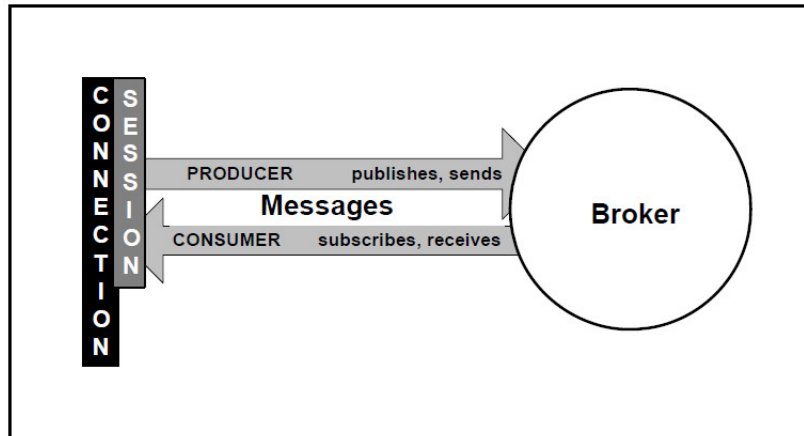
Transactions are scoped to **Session** objects. Starting in JMS 1.1, a transaction can include both Point-to-Point and Pub/Sub messages.

MessageConsumer and MessageProducer

A **Session** creates **MessageProducer** and **MessageConsumer** objects. The main responsibility of a **MessageProducer** object is to send messages from your client application to destinations on the broker. The main responsibility of a **MessageConsumer** object is to receive 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** are used to refer, respectively, to entities that receive and send messages. The following figure illustrates the roles of producers and consumers.

Figure 4: 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 *Aurea 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 of which extend the **Destination** interface). A **Destination** can be implemented as an administered object.

When you write an application using the JMS 1.1 common 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 [Messages](#) on page 191

Quality of Service and Protection

Some messages are simple and transitory, and they 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 as an alternative to quality.

All the SonicMQ message services and protection are available to 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—can be found in the following table:

Table 2: Services and Protection Available in SonicMQ Messaging

Service	Technique	Process	Reference
ENCRYPTED Content is encrypted.	Independent encryption mechanisms.	Body is appended after it has been encrypted, providing assurance that a message is protected even if the connection is insecure.	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 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 <i>Aurea SonicMQ Deployment Guide</i> 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 <code>NON_PERSISTENT_ASYNC</code> acknowledgement mode	

Service	Technique	Process	Reference
INTEGRITY When a destination has a QoS 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 (QoS) setting on destination is integrity. Also implicitly set when a sender chooses per-message encryption.	See the <i>Aurea SonicMQ Configuration and Management Guide</i> and the <i>Aurea SonicMQ Deployment Guide</i> for information about administrator settings for integrity and privacy. Note also information about how the installed cipher suites for QoS encryption can be customized.
PRIVACY When a destination has a QoS 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 (QoS) setting on destination is privacy or message producer explicitly requests privacy. Setting privacy includes the services of integrity.	The privacy setting can be explicitly requested by a message producer to a security-enabled broker. See Per Message Encryption on page 209.
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.	
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 169.
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 . See also Durable Subscriptions .

Service	Technique	Process	Reference
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 .
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 . See also Message Management by the Broker .
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 . See also Session Objects on page 173 and Reply-to Mechanisms .
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 <i>Aurea SonicMQ Deployment Guide</i> 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.	
ACKNOWLEDGED CONSUMPTION 1. Consumer acknowledges receipt to broker. 2. Client acknowledges receipt of received messages when session parameter is CLIENT_ACKNOWLEDGE or SINGLE_MESSAGE_ACKNOWLEDGE then when client calls acknowledge().	1. Acknowledgement type for the session was set when the session was created. 2. Explicit call by consumer.	1. Functions automatically to perform the specified type of acknowledgement for all messages consumed in that session. 2. Manual.	1. See Acknowledgement Mode on page 169.

Service	Technique	Process	Reference
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 would be replicated.	See Request and Reply Samples . See also Session Objects on page 173 and Reply-to Mechanisms .
DEAD MESSAGE QUEUE Sender/publisher can set properties to either or both re-enqueue undelivered messages and 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 208. See also the Dynamic Routing information in the <i>Aurea SonicMQ Deployment Guide</i> .

Clients

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

Java Client

SonicMQ clients are a set of Java archives that provide libraries of functionality that enable applets, proxy servers, servlet engines, and JavaBeans.

JMS Test Client

The SonicMQ JMS Test Client provides a graphical interface to demonstrate PTP and Pub/Sub messaging. You can use the JMS Test Client to send messages to queues and topics and to view the message properties and headers. See [Using the JMS Test Client](#) .

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 JMS 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 JMS properties from specified HTTP fields. See the *Aurea SonicMQ Deployment Guide* for more information and for samples of HTTP Direct.

Java Applet

SonicMQ can work in a Java applet running in a browser context to invoke classes that implement JMS functionality.

.NET Client

The SonicMQ .NET Client lets you write applications in a variety of Microsoft programming languages including C# .NET and Visual Basic .NET. The C# API enables interoperability between .NET applications and Java applications, thereby leveraging and extending the range of SonicMQ brokers. The C# Client includes features for fault tolerant connections, and transactional support. It is a native .NET component with fully-managed .NET code so that it works under the Microsoft CLR.

C/C++ Clients

SonicMQ can act as a pure C++ or pure ANSI C application on your system yet interface with a SonicMQ broker with the same behaviors as a pure JMS client. This provides legacy systems with integration and Web connection opportunities within the familiar operating characteristics of C and C++.

COM Client

SonicMQ provides a COM wrapper to the C++ client so that it can enable pure COM application on your system that interface with a SonicMQ broker with the same behaviors as a true JMS client. Examples are provided that demonstrate use of the COM client in Active Server Pages, Visual C++, Visual Basic, and VBScript applications.

SonicMQ API

The SonicMQ API provides Java and SonicMQ packages containing interfaces and methods you can use in your SonicMQ programming. The SonicMQ API documentation is located in your SonicMQ installation directory at `MQ_install_root\docs\sonicmq_api`. The SonicMQ API contains the following interfaces:

- Java Extension Package:
 - **javax.jms**
- SonicMQ Packages:
 - **progress.message.jclient** — Contains interfaces and classes used with SonicMQ
 - **progress.message.jclient.channel** — Contains the **RecoverableFileChannel** interface
 - **progress.message.xa** — Contains interfaces and classes used with XA Transactions
 - **com.sonicsw.stream** — Contains the **SonicStream** interface

Using the JMS Test Client

When you develop a messaging application, you want to be sure your messages have the correct content and are delivered as expected to the correct destinations. The SonicMQ JMS Test Client is a useful graphical tool that helps you do this. With this tool, you can create message producers (**QueueSenders** and **Publishers**) and message consumers (**QueueReceivers** and **Subscribers**); you can also create messages, send the messages to selected queues and topics, and visually inspect the messages after they are delivered. Many of the samples described in this book require you to use the JMS Test Client.

This chapter includes the following sections that describe how to use the JMS Test Client with the PTP and Pub/Sub messaging models.

- [Testing Point-to-point Messaging](#) on page 33
- [Testing Publish and Subscribe Messaging](#) on page 42

For details, see the following topics:

- [Testing Point-to-point Messaging](#)
- [Testing Publish and Subscribe Messaging](#)

Testing Point-to-point Messaging

Establishing a test PTP session with the JMS Test Client involves the tasks described in these sections:

- [Starting the SonicMQ® Container and Broker](#) on page 34
- [Establishing a Queue Session](#) on page 35

- [Creating Queue Senders and Queue Receivers](#) on page 36
- [Sending and Receiving Messages](#) on page 38
- [Browsing Messages on a Queue](#) on page 40

Starting the SonicMQ® Container and Broker

Be sure the SonicMQ® container and broker are running before executing any of the JMS Test Client samples. The following procedures explain how to start the SonicMQ container and broker on Windows, Linux, and UNIX platforms.

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

To start the broker and container from the Windows Start menu:

Select **Start > All Programs > Aurea > Sonic 2015 > Start Domain Manager** .

Starting the broker process

To start the broker process from a Linux or UNIX console window:

In a new command line console window, change directory to `<sonic_install_dir>/Containers/Domain1.DomainManager`, type **launchcontainer.sh** and press **Enter**.

Important: You can minimize the console window. Closing it, however, stops the Domain Manager.

Opening the JMS Test Client

You can open the JMS Test Client from the Windows start menu or LINUX/UNIX console window:

- [Starting the JMS Test Client from Windows](#) on page 35
- [Starting the JMS Test Client from Linux or UNIX](#) on page 35

Alternatively, you can open the **Sonic Management Console**, and select **Tools > JMS Test Client**.

(See [Starting the SonicMQ Container and Management Console](#) for information about starting the Management Console.)

The **JMS Test Client** windows opens.

Starting the JMS Test Client from Windows

To start the JMS Test Client from the Windows Start menu:

Select **Start > All Programs > Aurea > Sonic 2015 > Tools > JMS Test Client**

Starting the JMS Test Client from Linux or UNIX

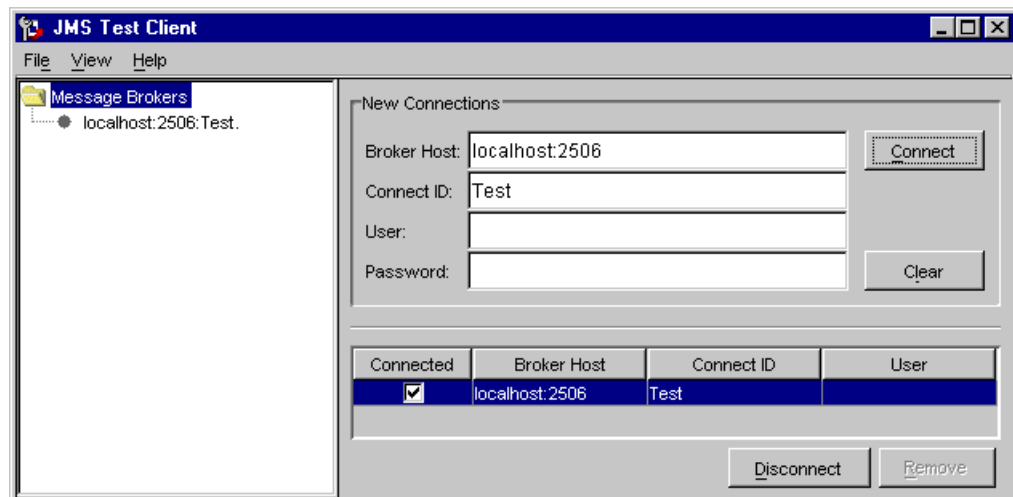
To start the JMS Test Client from a Linux or UNIX console window:

In a new command line console window, change directory to the `<SonicMQ install directory>/bin`, type the command `testClient.sh`.

Establishing Connection to the SonicMQ Broker

To connect the JMS Test Client to the broker:

1. In the **Broker Host** field, enter the information for your connection.
For example, **localhost:2506**.
2. In the **Connect ID** field, enter a unique name for your connection.
This example uses the **Connect IDTest**.
3. Click **Connect** to establish the connection, as shown in the following figure.



Establishing a Queue Session

The following procedure describes how to create a queue session in the JMS Test Client.

To create a queue session with the JMS Test Client:

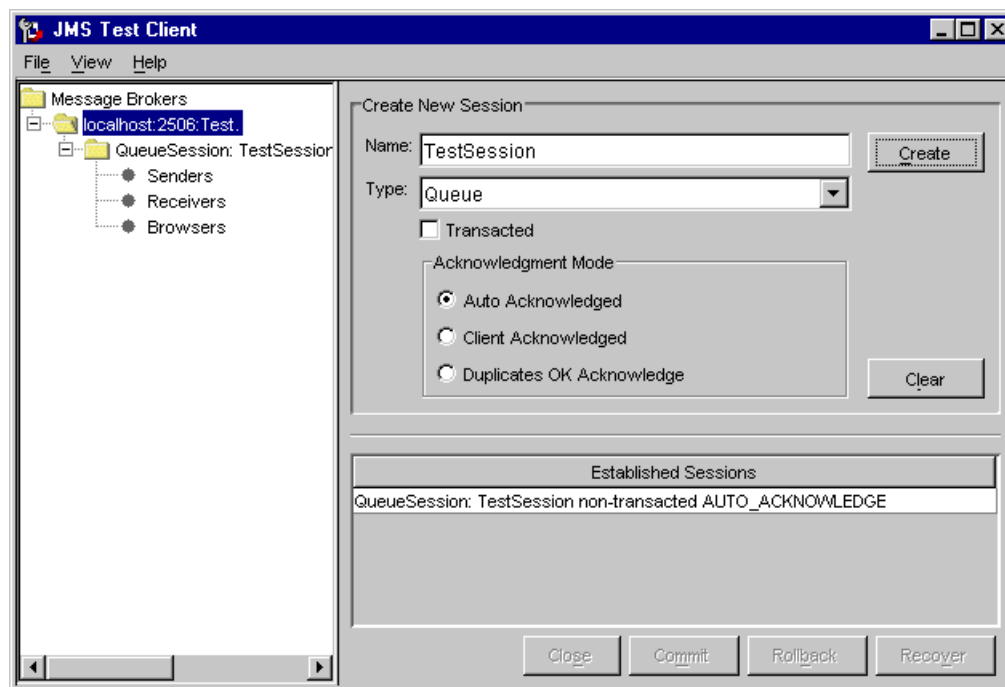
1. In the left panel of the **JMS Test Client** window, click the node for your message broker connection.
2. Type a unique string in the **Name** field and press **Enter**.

This example uses the name **TestSession**.

3. Select **Queue** from the **Type** drop-down list then click **Create**.

The session appears in the left panel with **Senders**, **Receivers**, and **Browsers** nodes as shown in the following figure:

Figure 5: Queue Session



Creating Queue Senders and Queue Receivers

The following procedure describes how to create queue senders and receivers in the JMS Test Client. You can only create senders and receivers to established queues. See the *Aurea SonicMQ Configuration and Management Guide* for information about creating and managing queues.

To create queue senders and receivers:

1. Select the **Senders** node in the left panel.

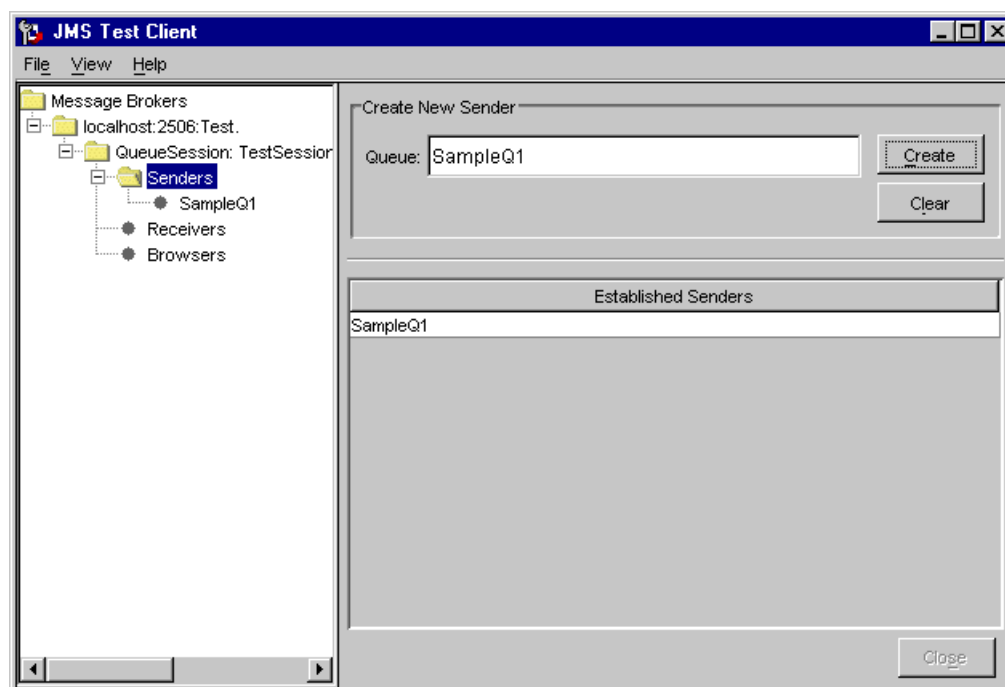
The right panel displays established senders that have been started from this session (if any) and allows you to create new senders.

2. To create a new sender, enter the name of the queue you want to send to in the **Queue** field and select **Create**.

A node for the new sender appears under the **Senders** node and the name of the queue appears in the **Established Senders** list.

The following figure creates a sender to the queue **SampleQ1**.

Figure 6: Create a Sender to SampleQ1



Note: You can only create a sender to an existing queue. See the *Aurea SonicMQ Configuration and Management Guide* for information about viewing existing queues and creating new queues.

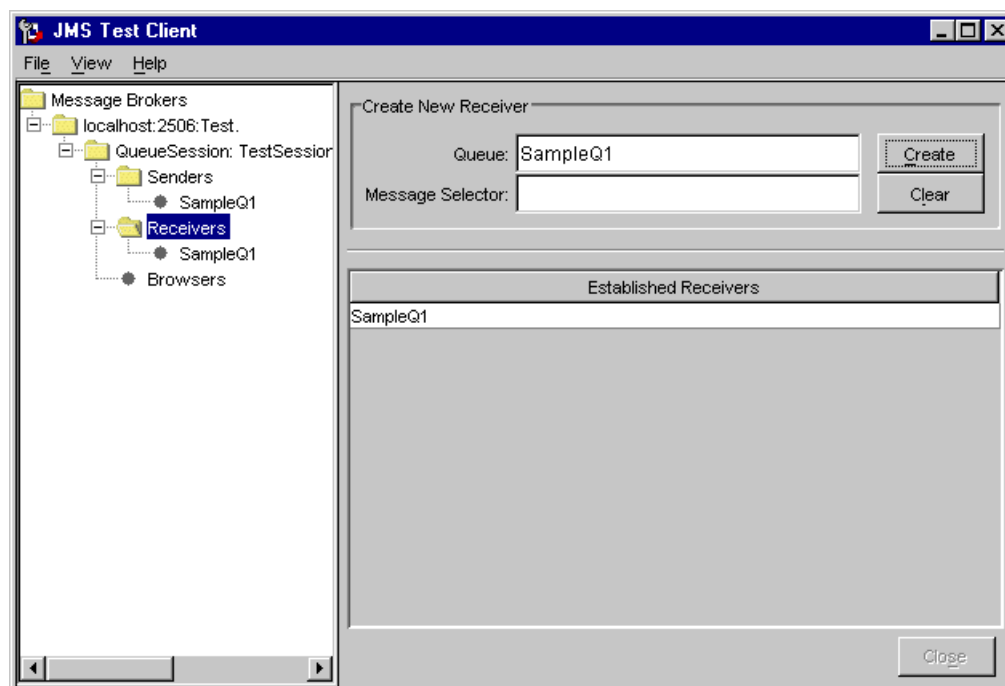
3. Select the **Receivers** node in the left panel.

The right panel displays the receivers that have been established in this session (if any) and allows you to create new receivers.

4. To create a new receiver:
 - In the **Queue** field, enter the name of the queue you from which want to receive messages.
 - This example creates a receiver to **SampleQ1**.
 - Optionally, you can use the **Message Selector** field to set up a query against header fields and properties to filter the available messages. See [Message Selection](#) for information about using message selectors.
 - This example does not use a message selector.
 - Click **Create**.

A node for the new receiver appears under the **Receivers** node and the name of the queue appears in the **Established Receivers** list, as shown in the following figure.

Figure 7: Create a Receiver for SampleQ1



Note: You can only create a receiver to an existing Queue. See the *Aurea SonicMQ Configuration and Management Guide* for information about viewing existing queues and creating new queues.

- To create additional receivers, select **Clear** and then repeat steps 3 and 4 of this procedure. With senders and receivers established, you can send and receive messages.

Sending and Receiving Messages

The following procedures describe how to send and receive messages on the queues for which you created senders and receivers in the preceding sections.

Note: Before continuing with this section, make sure you have completed the procedures in [Creating Queue Senders and Queue Receivers](#) on page 36.

Sending Messages

To send messages:

- Select a **Sender** in the left panel of the **JMS Test Client** window.

The right panel displays three tabs: **Header**, **Properties**, and **Body**. You can examine the default values under these tabs. In this example, you do not need to change any default settings or specify any message properties or body content.

- Select **Send** to send the message.

The next procedure shows you how to view the message just sent to **SampleQ1** on the receiver you created for that queue.

Viewing Received Messages

To view received messages:

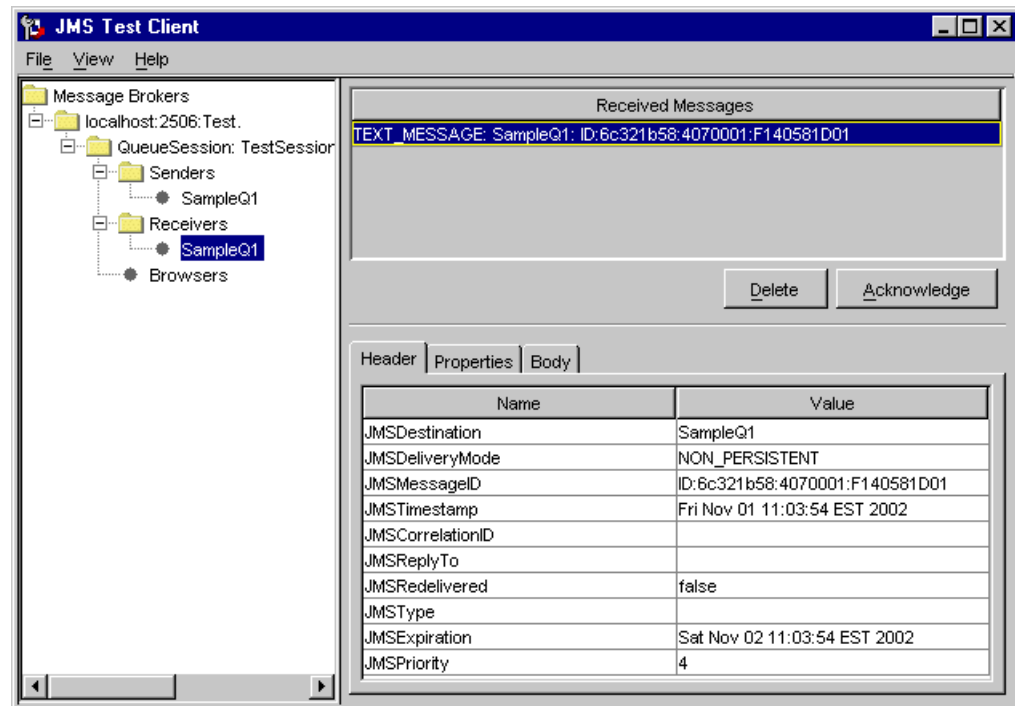
1. Under the **Receivers** node in the left panel, select the receiver for the queue to which you sent your message (in this example, **SampleQ1**).

The right panel displays the messages sent to this receiver. In this example, one message is displayed in the **Received Messages** area.

2. Select the message displayed in the **Received Messages** area.

The **Header**, **Properties**, and **Body** tabs in the lower right panel contain information for the received message, as shown in the following figure:

Figure 8: Received Messages



3. To delete one or more messages without acknowledging them, select the messages and click **Delete**.
4. To explicitly acknowledge one or more messages, select the messages and click **Acknowledge**.

An acknowledgement is sent back to the broker if the session was established in **Client Acknowledged** mode. (Messages can also be automatically acknowledged, depending on how the session was established.)

Note: By default, the number of viewable messages held in the **Received Messages** table is 50.

Browsing Messages on a Queue

The following procedure describes how to browse messages on a queue.

To browse messages on the queue:

1. Select the **Browsers** node in the left panel of the **JMS Test Client** window.

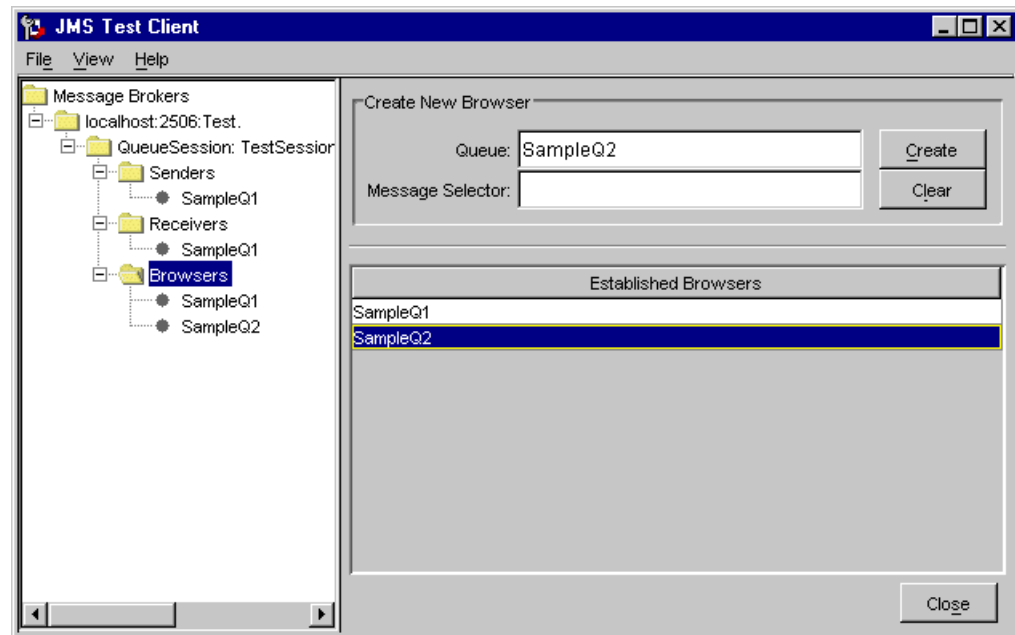
The right panel displays the queues available for browsing in the **Established Browsers** area. You can also create a browser for an existing queue.

This example includes the existing browser for **SampleQ1** (if you completed the preceding examples in [Creating Queue Senders and Queue Receivers](#) on page 36).

2. To create a new queue browser:
 - In the **Queue** field, enter the name of the queue where you want to browse messages.
 - This example creates a browser for **SampleQ2**.
 - Optionally, you can use the **Message Selector** field to set up a query against header fields and properties to filter the available messages. See [Message Selection](#) for information about using message selectors.
 - This example does not use a message selector.
 - Click **Create**.

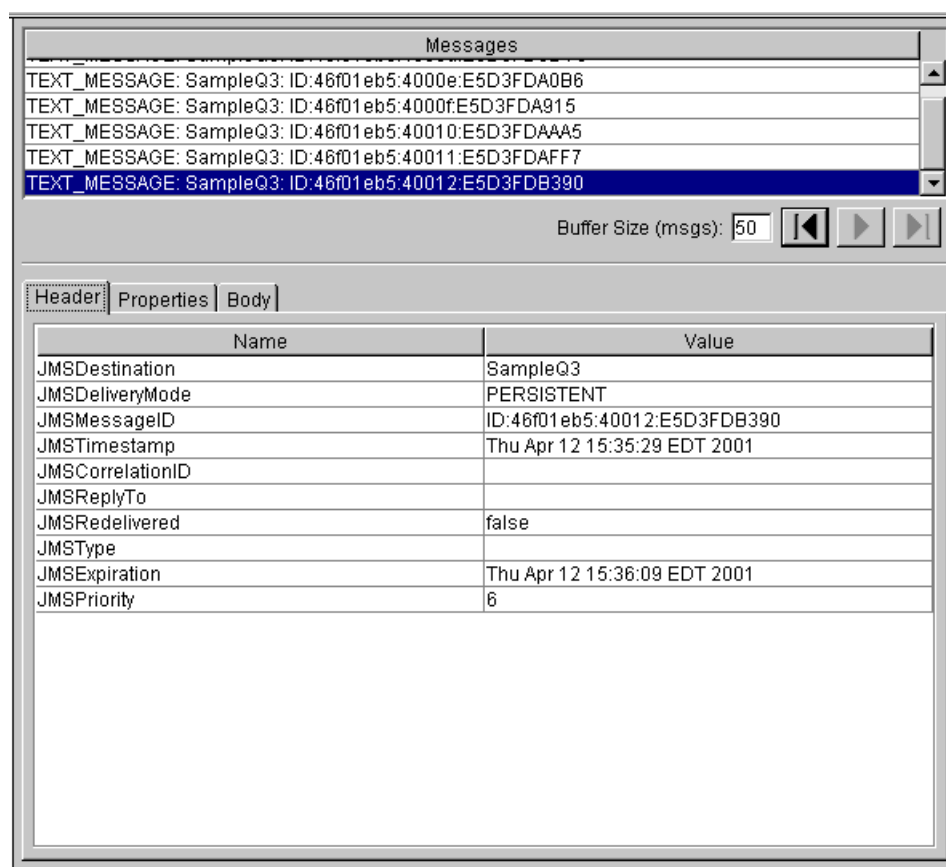
A node for the new queue browser appears under the **Browsers** node in the left panel, and the queue for the new browser appears in the **Established Browsers** list, as shown in the following figure:

Figure 9: Queue Browser Creation



3. Select the queue browser for the queue you want to view, and select the appropriate position-selector (black arrow heads) to position the browser cursor to the messages you want to see, as shown in the following figure:

Figure 10: Queue Browser



Note: To start browsing messages, you must first choose the left-most position-selector. You can restart the browse by choosing the same selector when the browser cursor is at any position in the queue. The two right-most position-selectors are active only when there are more messages on the queue than the specified buffer size. The move-forward position-selector shows the next buffer-size number of messages in the queue. The move-to-end position-selector shows the last buffer-size number of messages in the queue.

Testing Publish and Subscribe Messaging

You can use the JMS Test Client to simulate parts of your application and to demonstrate the behavior of various broker modes by establishing a test Publish and Subscribe session. To publish (or send) messages, a connection must be started. The connection starts automatically when there are adequate resources:

- Broker
- Connection to the broker
- Session on the connection
- Message mechanism (publisher, subscriber, listener, receiver, sender)

As you add, modify, or delete any resources, the connection automatically stops to allow the update and then restarts to allow publishing or sending of messages. Establishing a Pub/Sub session involves the tasks described in the following sections:

- [Establishing a Topic Session](#) on page 43
- [Creating Publishers and Subscribers to Topics](#) on page 44
- [Publishing Messages](#) on page 46
- [Receiving Messages on Subscribed Topics](#) on page 49

Establishing a Topic Session

The following procedure describes how to create a topic session.

To establish a topic session:

1. Open the **JMS Test Client**.

See [Opening the JMS Test Client](#) on page 34 for instructions.

2. In the left panel of the **JMS Test Client** window, click the node for your message broker connection.

3. Type any unique string in the **Name** field and press **Enter**.

This example uses the session name **TestSession**.

4. Select **Topic** from the **Type** drop-down list.

5. Check **Transacted** if you require a transacted session.

This example does not require a transacted session.

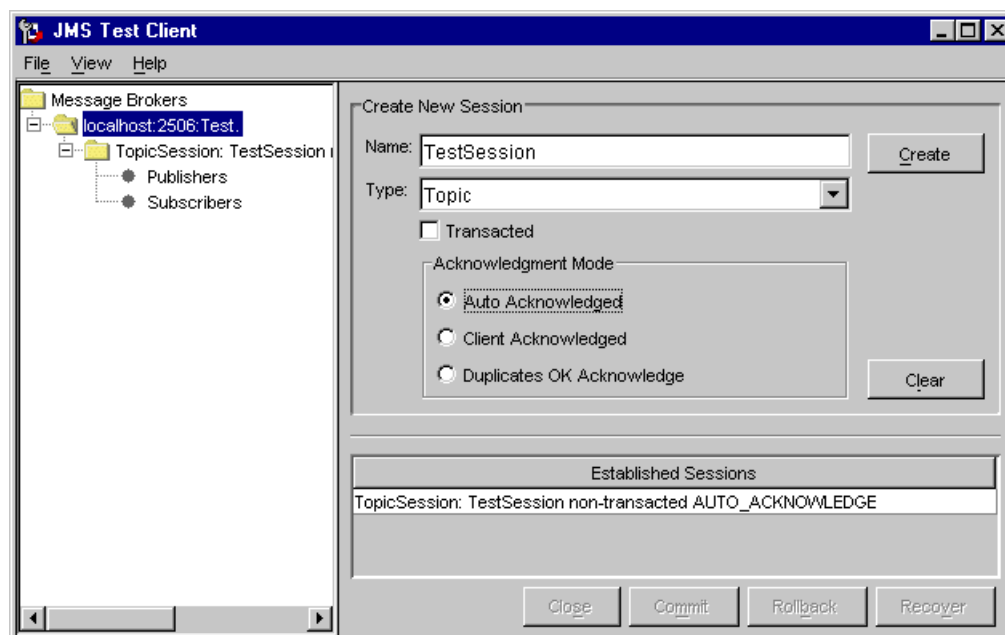
6. In the **Acknowledgement Mode** group, the **Auto Acknowledged** radio button is selected by default. You can change the mode to either **Client Acknowledged** or **Dups OK Acknowledge** (see [Acknowledgement Mode](#) on page 169 for information about the different acknowledgement modes).

This example uses the **Auto Acknowledged** mode.

7. Click **Create**.

The session appears in the left panel with **Publishers** and **Subscribers** nodes, as shown in the following figure:

Figure 11: Topic Session



Creating Publishers and Subscribers to Topics

The following procedure describes how to create publishers and subscribers to a topic using the JMS Test Client. You should complete the procedure in [Establishing a Topic Session](#) on page 43 before continuing with this section.

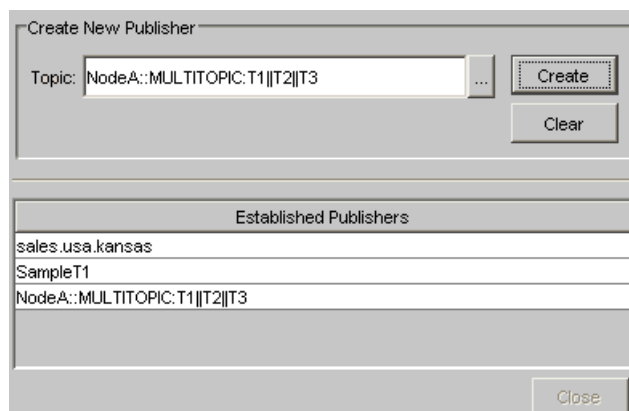
Creating a Publisher

To create a publisher:

1. Select the **Publishers** node in the left panel.

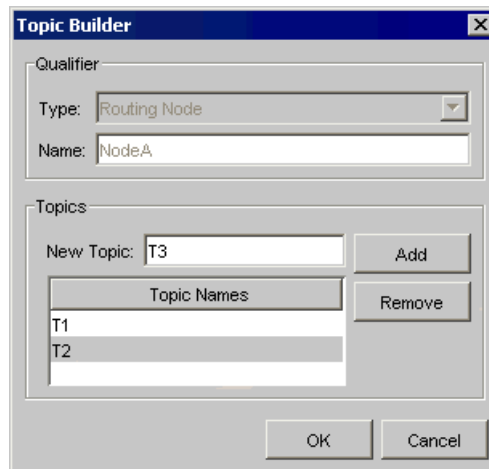
The right panel displays established publishers for this topic session (if any), and allows you to create and manage publishers, as shown in the following figure:

Figure 12: Publishers



The test client supports basic topics, hierarchical topics, node qualified topics, and MultiTopics as defined in the **Topic Builder** shown in the following figure.

Figure 13: Building MultiTopic Publishers



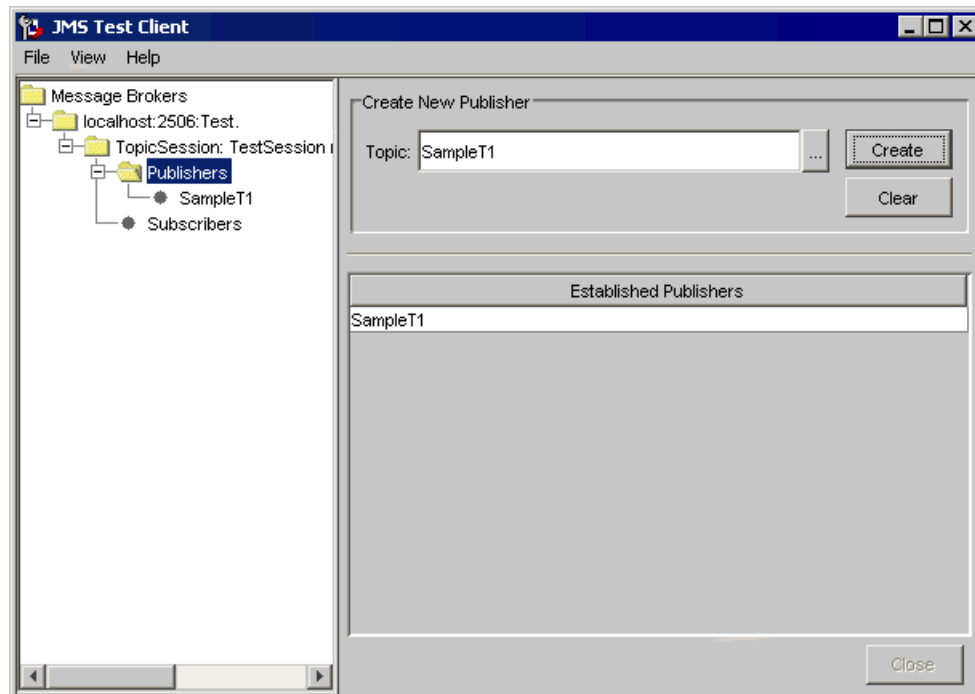
See [MultiTopics](#) for information about the syntax and behaviors of MultiTopic publishers and subscribers.

2. To create a new publisher, enter the name of the topic where messages are to be published in the **Topic** field and click **Create**.

A node for the new publisher appears under the Publishers node and the name of the connection.

This example creates a publisher to the topic **SampleT1**, as shown in the following figure:

Figure 14: Create a Publisher to SampleT1



Creating a Subscriber

To create a subscriber:

1. Select the **Subscribers** node in the left panel.

The right panel displays the **Established Subscribers** (if any) and allows you to create new subscribers.

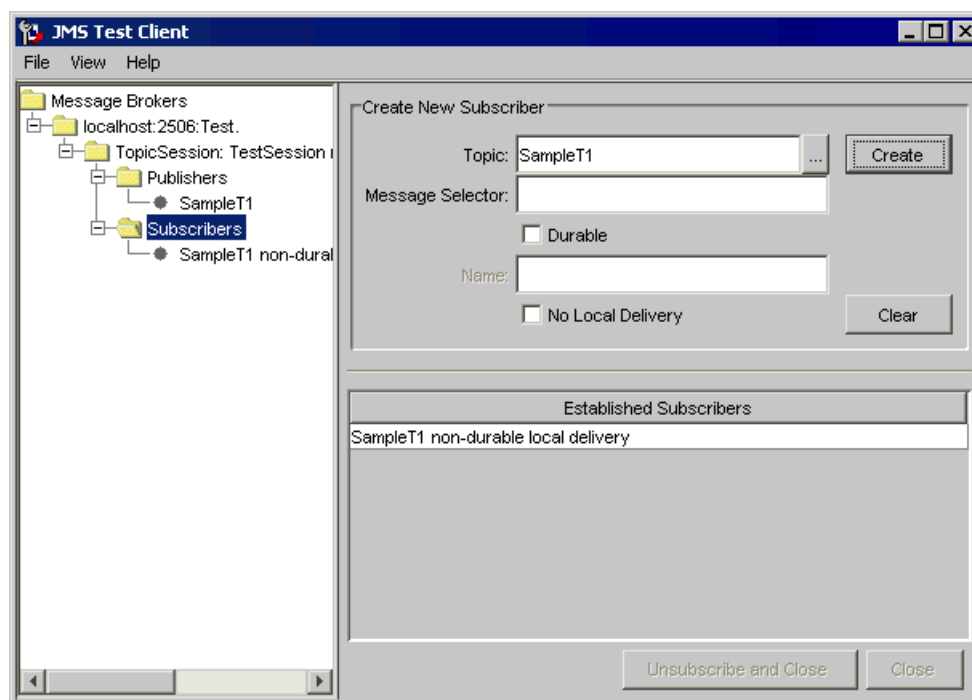
2. In the **Topic** field, enter the name of the topic to which you want to subscribe.

This example creates a subscriber to the topic **SampleT1**.

3. Optionally, you can use the **Message Selector** field to enter query values based on JMS header fields and properties to filter out unwanted messages. (For information on the syntax of this string, see [Messages](#) on page 191)
4. Optionally, you can create a durable subscription by checking **Durable** and entering a name in the **Name** field.
5. Optionally, you can check **No Local Delivery**, and the subscriber will not receive messages from publishers on the same connection.
6. Click **Create**.

The new subscriber appears under the **Subscribers** node in the left panel, and the topic appears in the list of **Established Subscribers** in the right panel, as shown in the following figure:

Figure 15: Create a Subscriber to SampleT1



See [Hierarchical Name Spaces](#) on page 345 for information about naming conventions.

With publishers and subscribers established, you can publish and subscribe to messages.

Publishing Messages

The following procedure describes how to publish messages.

Note: Before continuing with this section, make sure you have completed the procedures in [Creating Publishers and Subscribers to Topics](#) on page 44.

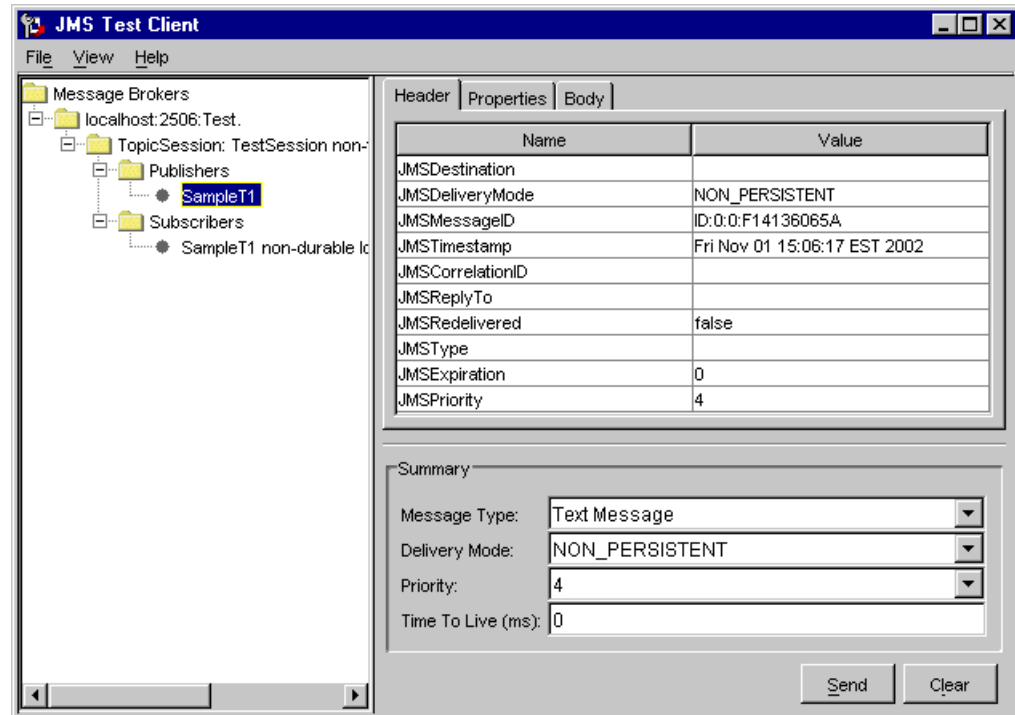
To publish messages:

1. Select a publisher in the left panel of the **JMS Test Client** window.

The right panel displays three tabs: **Header**, **Properties**, and **Body**. You can examine the default values under these tabs.

2. The **Header** tab, shown in the following figure, displays the header properties.

Figure 16: Message Header



You can edit only the following items in the header list:

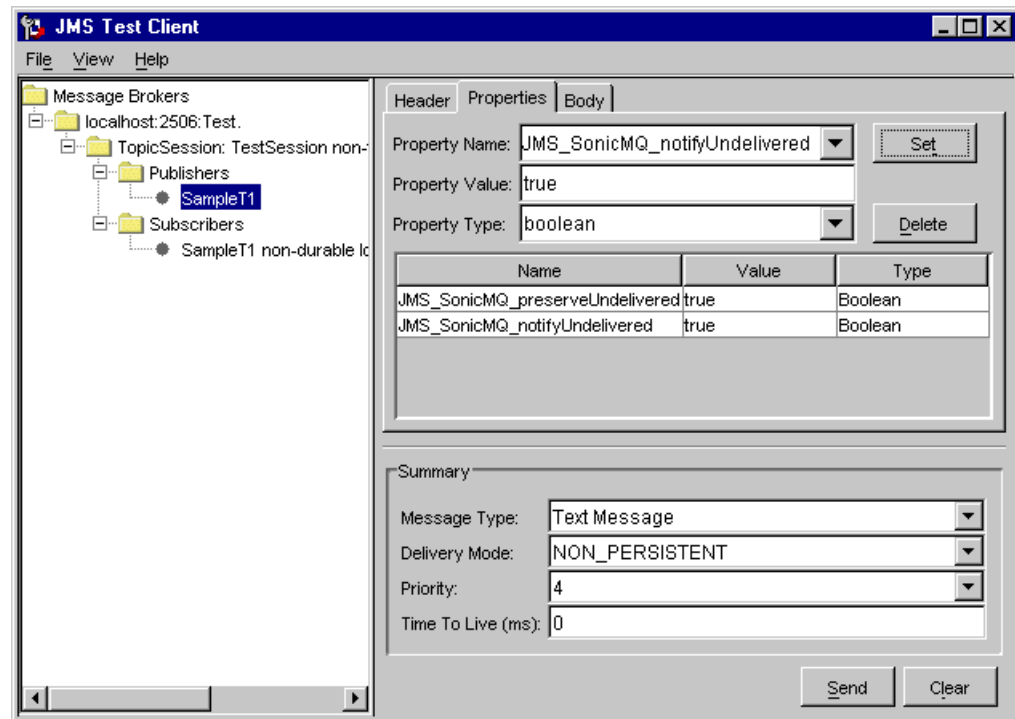
- JMSCorrelationID
- JMSReplyTo
- JMSType

In the **Summary** section, you can specify:

- **Message Type** — Message, Text Message, or XML Message
 - **Delivery Mode** — DISCARDABLE, NON_PERSISTENT, NON_PERSISTENT_REPLICATED, PERSISTENT
 - **Priority** — Integer values **0** (the highest) through **9** (the lowest)
 - **Time To Live** — In milliseconds, with **0** indicating no expiration
3. Select the **Properties** tab to define property values, as shown in the following figure, including the following SonicMQ-specific properties:
 - JMS_SonicMQ_preserveUndelivered

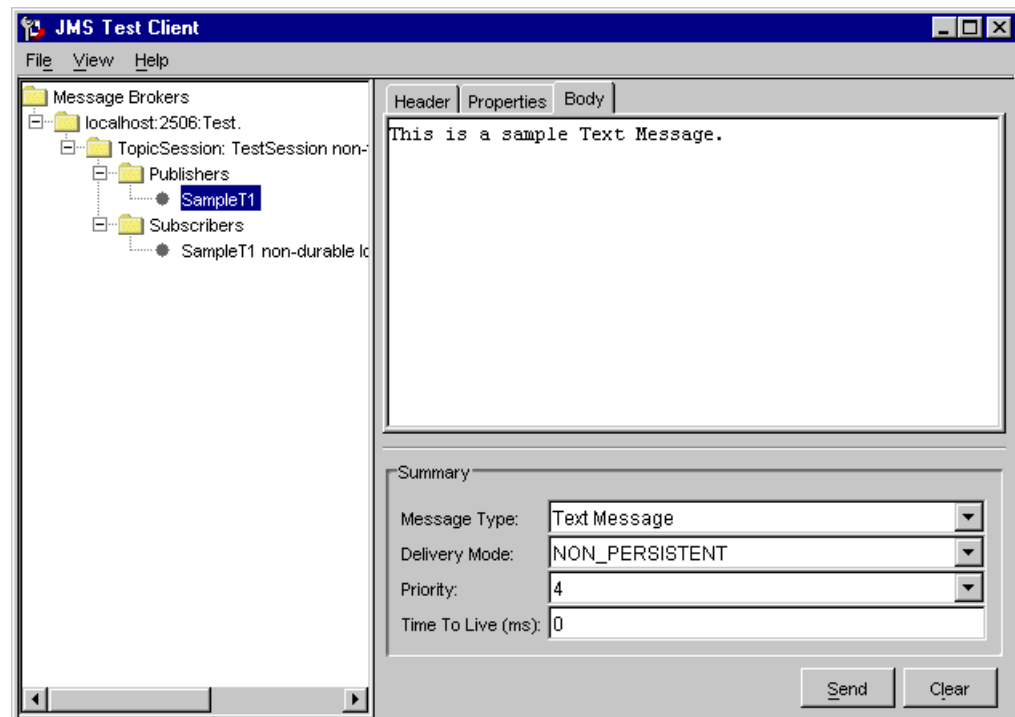
- JMS_SonicMQ_notifyUndelivered

Figure 17: Message Properties



4. Select the **Body** tab to compose the body of the message, as shown in the following figure.

Figure 18: Message Body



5. Select **Send** to send your message.

For information on message attributes, parameters, and properties, see [Messages](#) on page 191.

Receiving Messages on Subscribed Topics

The following procedure describes how to receive messages on a topic.

Note: Before continuing with this section, make sure you have completed the procedures in [Creating Publishers and Subscribers to Topics](#) on page 44 and [Publishing Messages](#) on page 46.

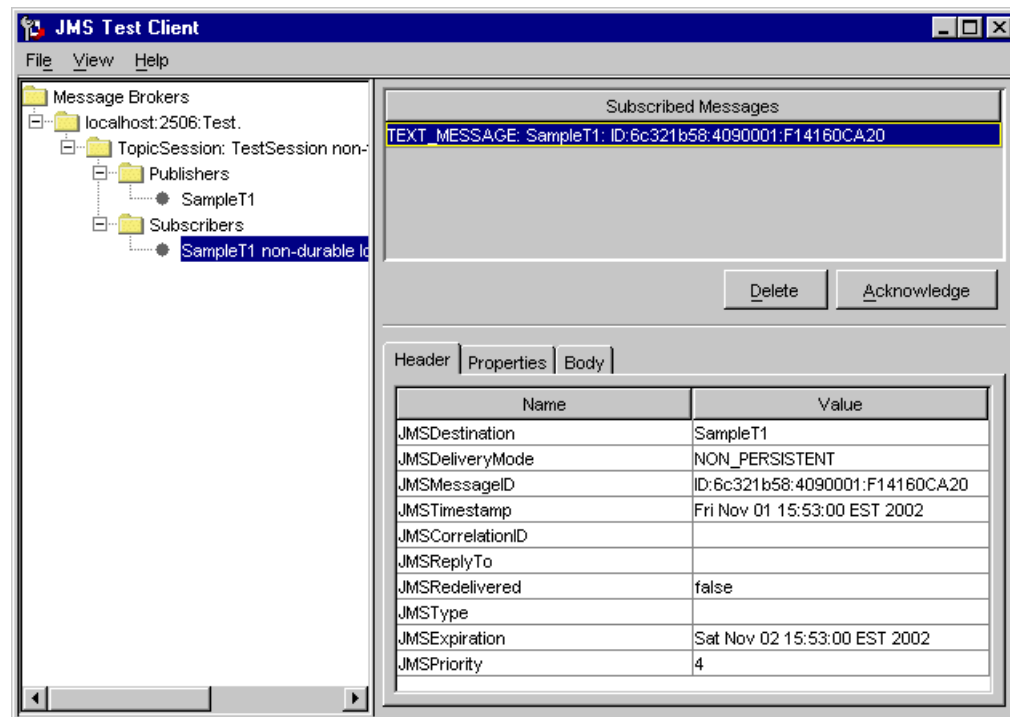
To receive messages on a topic:

1. Under the **Subscribers** node in the left panel, select the subscriber for the topic under which you published your message (in this example, **SampleT1**).

The right panel displays the messages on this subscriber. In this example, one message is displayed in the **Subscribed Messages** area.

2. Select the message displayed in the **Subscribed Messages** area. The **Header**, **Properties**, and **Body** tabs in the lower right panel contain information for the subscribed message, as shown in the following figure.

Figure 19: Subscribed Message



Note: By default, the number of viewable messages held in the Subscribed Messages table is 50.

3. To delete one or more messages without explicitly acknowledging them, select the messages and select **Delete**.
4. To acknowledge one or more messages, select the messages and select **Acknowledge**.

An acknowledgment is sent back to the broker if the session was established in **Client Acknowledged** mode.

Messages can also be automatically acknowledged, depending on how the session was established.

Examining the SonicMQ[®] JMS Samples

This chapter explains how to run the sample applications included with SonicMQ[®] software. These samples illustrate some of the SonicMQ[®] messaging functionality.

For details, see the following topics:

- [About SonicMQ® Samples](#)
- [Running the SonicMQ Samples](#)
- [Chat and Talk Samples](#)
- [MultiTopicChat Sample](#)
- [Samples of Additional Message Types](#)
- [Sample of Channels for Large Message Transfers](#)
- [Message Traffic Monitor Samples](#)
- [Transaction Samples](#)
- [Reliable, Persistent, and Durable Messaging Samples](#)
- [Request and Reply Samples](#)
- [Selection, Group, and Wild Card Samples](#)
- [Test Loop Sample](#)
- [Enhancing the Basic Samples](#)

About SonicMQ® Samples

The samples provided with SonicMQ® software are introduced in the *Getting Started with Aurea® SonicMQ®* manual. 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

Note: The samples in this chapter assume that you are using the default SonicMQ® setup, which does not enable security. Exercises are provided at the end of the chapter that detail how to reconfigure the persistent storage mechanism for security and how to enter the user names and passwords into the broker's authentication domain that security will demand. Without security, user names in the samples are arbitrary strings.

Important: [Connection Factories](#) on page 117 lists the characters that are not allowed in SonicMQ® names.

The SonicMQ® samples 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)
- **MultiTopic Chat** — This sample demonstrates how you can use MultiTopics to publish messages to multiple topics in a single operation and subscribe to multiple topics in a single subscription:
 - **MultiTopicChat** (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)
 - **XMLMessages** — Alternative parsers are used in both domains:
- **DOM2** — **XMLDOMTalk** (PTP), **XMLDOMChat** (Pub/Sub)
- **SAX** — **XMLSAXTalk** (PTP), **XMLSAXChat** (Pub/Sub)
- **Using Channels for Large Messages** — **FileSender**, **FileReceiver** (PTP)
- **Decomposing MultiPart Messages** — **Multipart** (PTP)
- **Message Traffic Monitors** — These samples provide views of message traffic in ways that are characteristic of their messaging domain:
 - **Messages on the Queue** — **QueueMonitor** (PTP)
 - **Messages to Subscribers** — **MessageMonitor** (Pub/Sub)
- **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. Persistent storage shows how the broker's PTP safety net, the Dead Message Queue, can trap undelivered messages. Durable subscription shows how a Pub/Sub subscriber can have messages held for them.

The samples in this category are:

- **Reliable Connection** — **ReliableTalk** (PTP), **ReliableChat** (Pub/Sub)
- **Persistent Storage** — **DeadMessages** (PTP)
- **Durable Subscription** — **DurableChat** (Pub/Sub)
- **Persistence on the Client**: **ContinuousSender**, **MessageReceiver** (PTP) and **ContinuousPublisher**, **MessageSubscriber** (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)
- **Selection, Grouping, and Wild Cards** — The message selector samples use SQL syntax to let the receiver 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 grouping provides the queue sender and the queue settings to direct assignment of receivers by group identifiers:
 - **Message Selection** — **SelectorTalk** (PTP), **SelectorChat** (Pub/Sub)
 - **Message Grouping** — **MessageGroupTalk** (PTP)
 - **Wild Cards** — **HierarchicalChat** (Pub/Sub)
- **Test Loop** — This sample shows how quickly messages can be sent and received in a test loop:
 - **Queue Test Loop** — **QueueRoundTrip** (PTP)

Other Samples Available

There are other SonicMQ samples available. As each requires a special setup to explore them, these samples are described in other chapters of this book, or in other SonicMQ documents:

- **Distributed Transactions** — The XA resources in SonicMQ provide the functionality to explore global transactions in a standalone sample. To see how to run these samples, see [Distributed Transactions Using XA Resources](#)
- **Dynamic Routing Queues** — When routing queues are established across brokers, messages are dynamic. The **GlobalTalk** (PTP) sample demonstrates dynamic routing queues in an appropriate setup. See the chapter “Multiple Nodes and Dynamic Routing” in the *Aurea SonicMQ Deployment Guide* for information about this sample.
- **SonicStreams API** — Using a special-purpose API, streams of indeterminate length can be transferred through a SonicMQ broker to multiple subscribers.
- **HTTP Direct** — These samples demonstrate ways to translate HTTP and HTTPS documents to JMS messages (inbound) and JMS messages to HTTP documents (outbound). The samples in this category are:
 - Basic Inbound
 - Basic Outbound
 - Basic Polling Receive
 - HTTP Direct for SOAP
 - HTTP Direct for JMS
 - HTTPS Authentication Samples

See the *Aurea SonicMQ Deployment Guide* for information about the HTTP Direct samples.

- **JNDI SPI** — Samples are provided to describe programming using the Sonic service provider implementation (SPI) for the Java Naming and Directory Interface (JNDI) See [Using the Sonic JNDI SPI](#) on page 371 for information.
- **Management Runtime and Configuration APIs** — Samples are provided to demonstrate the use of the SonicMQ Runtime and Configuration APIs. See the *Aurea SonicMQ Administrative Programming Guide* for information.
- **Replicated (High Availability) Brokers** — See the *Aurea 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](#) —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 *Aurea SonicMQ Deployment Guide* for complete SSL implementations you can explore; these implementations use the JSSE security software and credential samples installed with SonicMQ.
- **Security Enabled Dynamic Routing** — See the *Aurea SonicMQ Deployment Guide* for an example of how you can set up multiple brokers and security to realize secure dynamic routing across nodes.

Extending the Samples

After reviewing the sample applications, you can explore some variations:

- **Change the source files** — You can edit the source files, compile the changed file, and then run the applications again to observe the effect. Some ideas are presented as the following exercises:
 - Using a common destination for two different samples
 - Observing how different messaging behaviors affect round-trip times
 - Modifying the **MapMessage** to use other data types
 - Modifying the **XMLMessage** to show more data

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](#) provides an overview of the features and functions of security. But unless the broker chooses to enable 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 first having to 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 *Aurea SonicMQ Deployment Guide* for information about what you need to do to implement a SonicMQ sample in a secure environment.

Running the SonicMQ Samples

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

- [Starting the SonicMQ® Container and Management Console](#) on page 56
- [Opening Client Console Windows](#) on page 58
- [Using the Sample Scripts](#) on page 58

Starting the SonicMQ® Container and Management Console

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

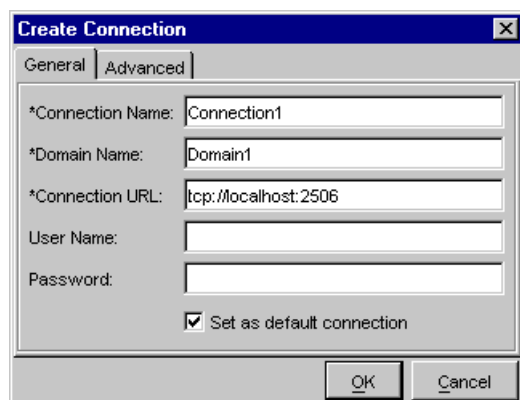
Note: If this is the first time you are running SonicMQ, you should not have to adjust the broker's settings. See the *Aurea® Sonic® Installation and Upgrade Guide* for more information.

Starting the broker process from the Windows Start menu

To start the broker process from the Windows Start menu:

1. Select **Start > All Programs > Aurea > Sonic 2015 > Start Domain Manager** .
SonicMQ starts the container that hosts the broker and then starts the broker.
2. Select **Start > All Programs > Aurea > Sonic 2015 > Sonic Management Console** .
SonicMQ opens the **Create Connection** dialog box, as shown in the following figure:

Figure 20: Create Connection



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.

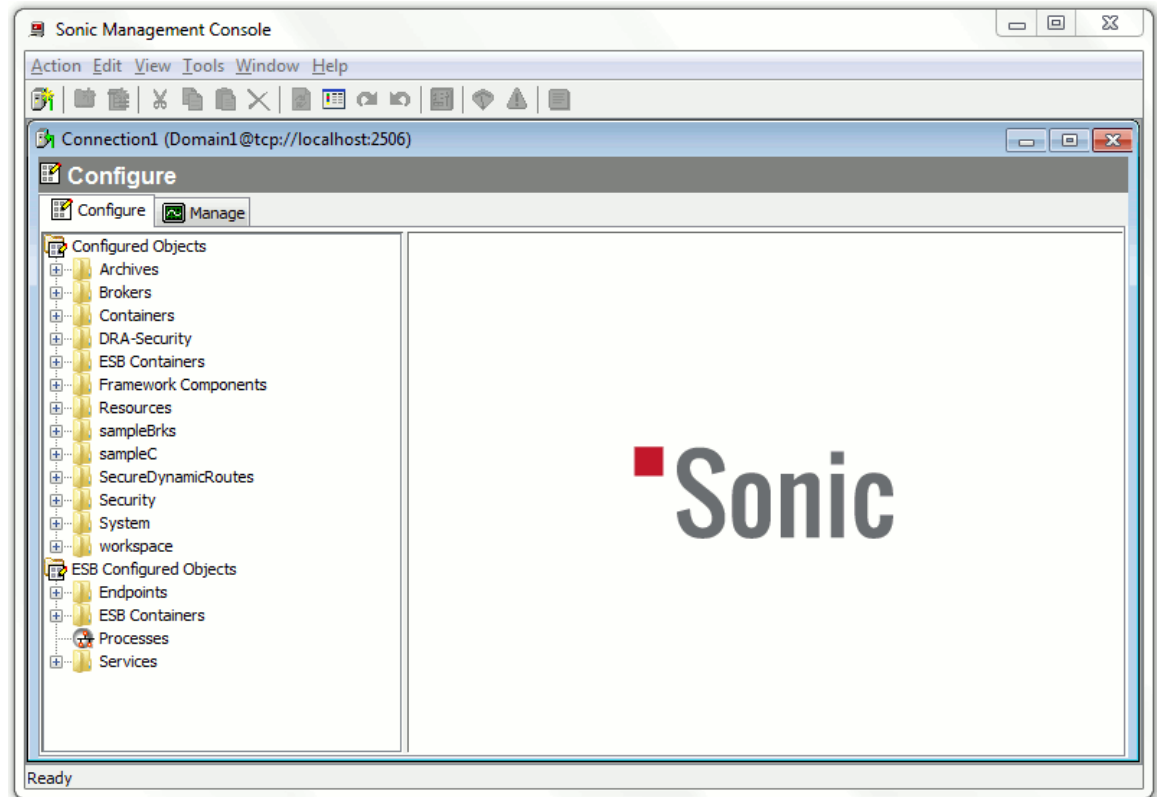
See the *Aurea SonicMQ Configuration and Management Guide* for information about setting parameters under the **Advanced** tab. For these samples, you do not have to set any advanced parameters.

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, as shown in the following figure.

Figure 21: Sonic Management Console Window



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

Starting the SonicMQ container and broker from a Linux or UNIX console window

To start the SonicMQ container and broker from a Linux or UNIX console window:

In a new command line console window, change directory to `<sonic_install_dir>/Containers/Domain1.DomainManager`, type `launchcontainer.sh` and press Enter.

The broker starts. The console window is dedicated to the process and, when running, displays:
SonicMQ Broker started, now accepting tcp connections on port 2506...

Important: You can minimize the console window. Closing the window, however, stops the broker.

The samples default to `localhost:2506`—a broker using port 2506 on the same system, **localhost**. If you use a different host or port, you need to specify the `host:port` parameter when you start each sample. For example:

```
..\..\sonicmq Chat -u User1-b hostname:2345
```

Opening the Sonic Management Console from a Linux or UNIX console window

To open the Sonic Management Console from a Linux or UNIX console window:

In a new command line console window, change directory to `<sonic_install_dir>10.0/MQ/bin`, type `startmc.sh` and press Enter. The Sonic Management Console opens.

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. There are conventions that you must follow depending on the platform:

- **Windows** — The scripts defer to Windows conventions.
- **Linux and UNIX platforms** — Instead of using `.bat` files, use the `.sh` file at the same location. Substitute forward slash (/) wherever back slash (\) is used as a path delimiter. Any sourcing is handled in the shell scripts.

Note: Consider all text to be case-sensitive. While there might be some platforms and names where case is not distinguished, it is good practice to always use case consistently.

Using the Sample Scripts

A universal script handler is installed at the **Samples** directory level. This script, **SonicMQ.bat** (`.sh` under Linux® and UNIX®), does the following:

- Points to the Java executable used by SonicMQ
- Sets the **CLASSPATH** for the SonicMQ `.jar` files as required
- Invokes the executable, its parameters, and a list of variables

The script is suitable for the basic samples provided, but you might have to adjust it if you use long parameter lists. Standard invocation of the script from a sample folder is two levels down.

Important: When you modify the original sample files, you can use the techniques described above to set up a universal compiler script. Replicate and modify **SonicMQ.bat** (`.sh` under Linux® and UNIX®) to something like **SonicMQ_javac.bat** (`.sh` under Linux® and UNIX®) and then confirm that **javac.exe** (or the path to your preferred compiler) is in the script.

Using the SonicMQ Samples in a Sonic® Workbench™ Installation

The default setting for security on the management broker in a Sonic® Workbench™ install is to enable security. That means that usernames in the samples intended to provide information to keep track of multiple application instances must be valid users. As the only default user is **Administrator** (with the password **Administrator**), you must either define the other users, or use the default user **Administrator** and the password (**-p**) parameter and the user's password on every command line. For example:

```
..\..\SonicMQ Chat -u Administrator -p Administrator
```

Using the SonicMQ Samples with a non-default Broker

If you want to use a broker on a remote system or different port for the samples, you have to specify it in the command line. The samples default to **localhost:2506**—a broker using port 2506 on the local system. If you use a different host or port, specify the broker parameter (**-b**) when you start each sample. For example:

```
..\..\SonicMQ Chat -u Market_Maker -b Eagle:2345
```

In summary, if you were on a Linux® system where a Sonic® Workbench™ is installed, and you want to connect to another system's messaging broker, you might enter:

```
../../SonicMQ.sh Chat -u Administrator -p Administrator -b Eagle:2345
```

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. In a new command line console window, change directory to **TopicPubSub\Chat** folder, then enter: `..\..\SonicMQ Chat -u Chatter1`

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

2. Open another command line console window, change directory to **TopicPubSub\Chat** folder, then enter: `..\..\SonicMQ Chat -u Chatter2`

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

3. In one of the **Chat** shells, 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.
4. In the other **Chat** shell, 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. In a new command line console window, change directory to `QueuePTP\Talk` folder, then enter: `..\..\SonicMQ Talk -u Talker1 -qr SampleQ1 -qs SampleQ2`

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

2. Open another command line console window, change directory to `QueuePTP\Talk` folder, then enter: `..\..\SonicMQ Talk -u Talker2 -qr SampleQ2 -qs SampleQ1`

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

3. 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.

4. 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 will display 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, every subscriber receives everyone else's messages but not their 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 the receivers take turns receiving the messages.

MultiTopicChat Sample

This sample demonstrates how an application can publish to multiple topics in a single operation using a MultiTopic. It also demonstrates how an application can subscribe to many topics in a single operation by using MultiTopic.

Setting Up MultiTopic Sessions

Before you can run the **MultiTopicChat** sample, you must start sessions as described below.

Starting MultiTopic sessions for publishing

To start MultiTopic sessions for publishing:

1. In a new command line console window, change directory to `TopicPubSub\MultiTopicChat` folder, then enter: `..\..\SonicMQ MultiTopicChat -u SALES`

This command starts a **MultiTopic** session for the user **SALES**. This user subscribes to the **jms.samples.chat.SALES** topic.

2. Open another command line console window, change directory to `TopicPubSub\MultiTopicChat` folder, then enter: `..\..\SonicMQ MultiTopicChat -u MARKETING`

This command starts a **MultiTopic** session for the user **MARKETING**. This user subscribes to the **jms.samples.chat.MARKETING** topic.

3. Open another command line console window, change directory to `TopicPubSub\MultiTopicChat` folder, then enter: `..\..\SonicMQ MultiTopicChat -u SUPPORT`

This command starts a **MultiTopic** session for the user **SUPPORT**. This user subscribes to the **jms.samples.chat.SUPPORT** topic.

Starting a MultiTopic session for subscribing

To start a MultiTopic session for subscribing:

Open a command line console window, change directory to `TopicPubSub\MultiTopicChat` folder, then enter: `..\..\SonicMQ MultiTopicChat -u AUDIT -l MARKETING,SUPPORT`

This command starts a **MultiTopic** session for the user **AUDIT**. This user subscribes to the **MARKETING** and **SUPPORT** topics.

Note: Using wildcards in the subscriber's list parameter — You can use the template characters pound (#) and asterisk (*) for subscriptions, for example, `-l #`. When using the list parameter (`-l`) with an asterisk, your shell might require you to enclose the asterisk in quotes: `-l "*"`.

Demonstrating MultiTopic Publish and Subscribe

This section describes how to use the MultiTopic sessions for publishing and subscribing.

Demonstrating MultiTopic publishing

To demonstrate MultiTopic publishing:

1. Choose a publishing session (**SALES**, **MARKETING**, or **SUPPORT**) and enter some text (for example, **Hello**) on the command line, then press **Enter**.
2. In the session you chose in step 1, enter a comma-separated list of the user names to which you want to send the message. For example, from the **SALES** session, enter:
`MARKETING, SUPPORT`

This causes the **SALES** session to publish the message to a MultiTopic consisting of the **jms.samples.chat.MARKETING** and **jms.samples.chat.SUPPORT** topics. The **MARKETING** and **SUPPORT** sessions receive the message on the topic subscribed.

Demonstrating MultiTopic subscribing

To demonstrate MultiTopic subscribing:

In the **SUPPORT** session:

- a) Enter some text on the command line, then press **Enter**.
- b) Enter the name of the user to send the message: **SALES**

Notice that the **AUDIT** session does not receive the message.

- c) Enter some text on the command line, then press **Enter**.
- d) Enter: `SUPPORT, MARKETING, SALES`

Notice that the **AUDIT** session receives one copy of the message.

Demonstrating split delivery of MultiTopic messages

To demonstrate split delivery of MultiTopic messages:

1. In the **AUDIT** session window, press **Ctrl+C** to stop the application.
2. Add the **-s** parameter to the command, and then restart the **AUDIT** session:

```
..\SonicMQ MultiTopicChat -u AUDIT -I MARKETING,SUPPORT -s
```

This command starts a split delivery **MultiTopic** session for the user **AUDIT**.

3. Enter some text on the command line, then press **Enter**.
4. Enter: `SUPPORT, MARKETING`

Notice that the **AUDIT** session receives two copies of the message, one for each topic in the MultiTopic list.

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 Java 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")
```

Starting MapTalk sessions

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 command line console window, change directory to `QueuePTP\MapTalk` folder, then enter: `..\..\SonicMQ MapTalk -u QAccounting -qr SampleQ1 -qs SampleQ2`

This command starts a **MapTalk** session for the user **QAccounting**.

2. Open another command line console window, change directory to the `QueuePTP\MapTalk` folder then enter: `..\..\SonicMQ MapTalk -u QAuditing -qr SampleQ2 -qs SampleQ1`

This command starts a **MapTalk** session for the user **QAuditing**.

Sending and receiving MapMessages

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 following source code of the sample **MapTalk.java**:

```
javax.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 following source code of the sample **MapTalk.java**:

```
String sender = mapMessage.getString("sender");
String content = mapMessage.getString("content");
System.out.println(sender + ": " + content);
```

XML Messages

XML data definitions with tagged text are useful for communicating structured sets of defined data records or transacted message sets over the Internet. The XML parser included with SonicMQ, the Apache Xerces XML Parser, interprets the data using Document Object Model (DOM) Element nodes. The message receiver window echoes its translation of the XML-tagged code derived from your text entry. For example, if you (as the sender **Catalog_Update**) enter **Item One**, the XML-tagged code is packaged as shown in the following sample code, an excerpt of the sample file **XMLDOMChat.java**.

XMLDOMChat.java: XML-Tagged Code

```
{
progress.message.jclient.XMLMessage xMsg =
((progress.message.jclient.Session) pubSession).createXMLMessage();
StringBuffer msg = new StringBuffer();
msg.append("<?xml version=\"1.0\"?>\n");
msg.append("<message>\n");
msg.append("  <sender>" + username + "</sender>\n");
msg.append("  <content>" + content + "</content>\n");
msg.append("</message>\n");
xMsg.setText(msg.toString());
publisher.send(xMsg);
}
```

The tagged message text is well-formed XML, as shown:

```
<?xml version="1.0"?>
<message>
  <sender>sender</sender>
  <content>message_content</content>
</message>
```

In the DOM samples, when the message is received, the embedded DOM2 XML parser is invoked. The message is interpreted to display the DOM nodes, as shown:

```
[XML from 'DOMSend'] Hello
ELEMENT: message
|--NEWLINE
+---ELEMENT: sender
|   |--TEXT_NODE: DOMSend
|--NEWLINE
+---ELEMENT: content
|   |--TEXT_NODE: Hello
|--NEWLINEXML DOM2 Messages (PTP)
```

In the SAX samples, when the message is received, the embedded SAX XML parser is invoked. The message is interpreted to display the message in XML format, as shown:

```
<?xml version="1.0"?>
<message>
  <sender>SAXSend</sender>
  <content>Bonjour</content>
</message>
```


XMLDOMTalk (PTP)

In this example, the first **XMLDOMTalk** session sends on the first queue and receives to the second queue while the other session does the opposite.

Starting PTP XMLDOMTalk sessions

To start PTP XMLDOMTalk sessions:

1. Open a command line console window, change directory to the `QueuePTP\XMLDOMTalk` folder, then enter: `..\..\SonicMQ XMLDOMTalk -u DOMSend -qr SampleQ2 -qs SampleQ1`

This command starts a **XMLDOMTalk** session for the user **DOMSend**.

2. Open another command line console window, change directory to the `QueuePTP\XMLDOMTalk` folder, then enter: `..\..\SonicMQ XMLDOMTalk -u DOMRecv -qr SampleQ1 -qs SampleQ2`

This command starts a **XMLDOMTalk** session for the user **DOMRecv**.

Sending and receiving PTP DOM2 XMLMessages

To send and receive PTP DOM2 XMLMessages:

In the **DOMSend** window, type text such as **Hello** and then press **Enter**.

The message appears in the **DOMRecv** window formatted in DOM2 nodes, as shown:

```
[XML from 'DOMSend'] Hello
ELEMENT: message
|--NEWLINE
+--ELEMENT: sender
|--TEXT_NODE: DOMSend
|--NEWLINE
+--ELEMENT: content
|--TEXT_NODE: Hello
|--NEWLINE
```

XMLSAXTalk (PTP)

In this example, the first **XMLSAXTalk** session sends on the first queue and receives to the second queue while the other session does the opposite.

Starting PTP XMLSAXTalk sessions

To start PTP XMLSAXTalk sessions:

1. Open a command line console window, change directory to the `QueuePTP\XMLSAXTalk` folder, then enter: `..\..\SonicMQ XMLSAXTalk -u SAXSend -qr SampleQ2 -qs SampleQ1`

This command starts a **XMLSAXTalk** session for the user **SAXSend**.

2. Open another command line console window, change directory to the `QueuePTP\XMLSAXTalk` folder, then enter: `..\..\SonicMQ XMLSAXTalk -u SAXRecv -qr SampleQ1 -qs SampleQ2`

This command starts a **XMLSAXTalk** session for the user **SAXRecv**.

Sending and receiving PTP SAX XMLMessages

To send and receive PTP SAX XMLMessages:

In the **SAXSend** window, type text such as **Bonjour** and then press **Enter**.

The message appears in the **SAXRecv** window in XML format, as shown:

```
<?xml version="1.0"?>
<message>
<sender>SAXSend</sender>
<content>Bonjour</content>
</message>
```

XMLDOMChat (Pub/Sub)

In this example, the **XMLDOMChat** sessions publish and subscribe on the topic **jms.samples.chat**.

Starting PTP XMLDOMChat sessions

To start PTP XMLDOMChat sessions:

1. Open a command line console window, change directory to the `TopicPubSub\XMLDOMChat` folder, then enter: `..\..\SonicMQ XMLDOMChat -u DOMPub`

This command starts a **XMLDOMChat** session for the user **DOMPub**.

2. Open another command line console window, change directory to the `TopicPubSub\XMLDOMChat` folder, then enter: `..\..\SonicMQ XMLDOMChat -u DOMSub`

This command starts a **XMLDOMChat** session for the user **DOMSub**.

Sending and receiving Pub/Sub DOM XMLMessages

To send and receive Pub/Sub DOM XMLMessages:

In the **DOMPub** window, type text such as **Bonjour** and then press **Enter**.

The message appears in the **DOMSub** window formatted in DOM2 nodes, as shown:

```
[XML from 'DOMPub'] Bonjour
ELEMENT: message
|--NEWLINE
+--ELEMENT: sender
|--TEXT_NODE: DOMPub
|--NEWLINE
+--ELEMENT: content
|--TEXT_NODE: Bonjour
|--NEWLINE
```

XMLSAXChat (Pub/Sub)

In this example, the **XMLSAXChat** sessions publish and subscribe on the topic **jms.samples.chat**.

Starting PTP XMLSAXChat sessions

To start PTP XMLSAXChat sessions:

1. Open a command line console window, change directory to the `TopicPubSub\XMLSAXChat` folder, then enter: `..\..\SonicMQ XMLSAXChat -u SAXPub`

This command starts a **XMLSAXChat** session for the user **SAXPub**.

2. Open another command line console window, change directory to the TopicPubSub\XMLSAXChat folder, then enter: `..\..\SonicMQ XMLSAXChat -u SAXSub`

This command starts a **XMLSAXChat** session for the user **SAXSub**.

Sending and receiving Pub/Sub SAX XMLMessages

To send and receive Pub/Sub SAX XMLMessages:

In the **SAXPub** window, type text such as **Hello** and then press **Enter**.

The message appears in the **SAXSub** window formatted in SAX format, as shown:

```
<?xml version="1.0"?>
<message>
<sender>SAXPub</sender>
<content>Hello</content>
</message>
```

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.

This sample composes a four-part message using JMS message types and data handlers. 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 command line console window, change directory to QueuePTP\MultipartMessage\XMLSAXChat folder, then enter: `..\..\SonicMQ 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
sending part2..some bytes
sending part3..a simple text string
sending part4..a Readme file
```

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, 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 = 4
-----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
-----Beginning of part 2
```

```
Part.contentType = myBytes
Part.contentId = CONTENTID2
...size : 38
...content :
This string is sending as a byte array
-----end of part 2
-----Beginning of part 2
Part.contentType = myBytes
Part.contentId = CONTENTID2
...size : 38
...content :
This string is sending as a byte array
-----end of part 2
-----Beginning of part 3
Part.contentType = text/plain
Part.contentId = CONTENTID3
...size : 37
...content :
a simple text string to put in part 3
-----end of part 3
```

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 **instanceof** operator to identify and cast the message into an **XMLMessage** or a **MapMessage**.

You can modify the source code of these samples to:

- Create a table of XML data that forms an **XMLMessage**.
- Set some map values to Java 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 100 that describe these changes. See also [Message Type](#) on page 192.

Sample of Channels for Large Message Transfers

Note: This sample requires a SonicMQ installation includes the ClientPlus libraries for the SonicMQ client.

SonicMQ with the ClientPlus option provides large message support by allowing a JMS message to be associated with an instance of a recoverable channel. The file that will be transferred will move through the recoverable channel dedicated to the sender. Internally, the file is sent in fragments. Fragment loss or duplication due to failure is handled internally.

To transfer a large message:

1. Identify a file that you want to transfer and note its absolute path.

For example, you can transfer the **.pdf** file for this book, located at:
MQ_install_root\docs\program.pdf

2. Identify or create a folder where a transferred file will be placed.

For example, *c:\Inbound*.

3. Open a command line console window, change directory to the *ClientPlus\LargeMessageSupport* folder then enter the following command as a single line: `..\..\SonicMQ FileReceiver -u aReceiver -p passwordRecv -qr SampleQ1 -d c:\Inbound`

A folder is created with the name **aReceiver** in the *ClientPlus\LargeMessageSupport* folder. This folder records the receiver status information about the channel and the inbound files that will be reconstructed in the directory, *c:\Inbound*.

Important: The **FileReceiver** starts before the **FileSender** because the receiver blocks indefinitely while the **FileSender** times out after 30 seconds then close its connection and exits. You can restart **FileSender** to complete the sample

4. Open another window to the *ClientPlus\LargeMessageSupport* folder then enter the following command as a single line: `..\..\SonicMQ FileSender -u aSender -p passwordSend -qs SampleQ1 -f MQ_install_root\docs\program.pdf`

A folder is created with the name **aSender** to record the file location and the recoverable channel data for the receiver of the message on **SampleQ1**.

The progress of the message transfer is displayed in the **FileSender** console window shown:

```
MQ_install_root\samples\ClientPlus\LargeMessageSupport>
..\..\SonicMQ FileSender -u aSender -p passwordSend -qs SampleQ1
-f MQ_install_root\docs\program.pdf
Session is created
Try to send header message and establish channel to send file -
MQ_install_root\docs\program.pdf
13021815223128Broker1 channel established!
File size to send - 2345468
....10%....20%....30%....40%....50%....60%....70%....80%....90%....100%
Transfer is complete!
Close connection and exit
```

When the message transfer is complete, the source file remains intact and the received file resides in the target directory. The **aSender** and **aReceiver** Channel folders are empty.

If the transfer failed in progress—either the sender or the receiver—the recoverable file channel info is accessible to both the sender and the receiver applications when they re-establish the channel through the broker.

Reviewing the Large Message Transfer Sample

The large message transfer sample shows:

- File transfer is, like FTP, a static physical file identified on one system replicated in a specified location on another system.
- File transfers can be interrupted and contain the logic and records to resume an interrupted transfer.
- The message that identifies the impending transfer is not needed for recovery. Recovery is defined in logs and resumes by assessing unfinished channels and continuing that defined transfer.

Message Traffic Monitor Samples

These samples each open GUI windows that provide a scrolling array of its contents. The nature of the two monitors underscores fundamental differences between the Publish and Subscribe messaging model and the Point-to-point messaging model. The following table shows these differences.

Table 3: Differences Between QueueMonitor and MessageMonitor

	QueueMonitor	MessageMonitor
What messages are displayed?	Undelivered.	Delivered.
When does the display update?	When you click the Browse Queues button, the list is refreshed.	When a message is published to a subscribed topic, it is added to the displayed list.
When does the message go away?	When the message is delivered (or when it expires).	When the display is cleared for any reason.
What happens when the broker and monitor are restarted?	Listed messages marked PERSISTENT are stored in the broker persistent storage mechanism. They are redisplayed when the broker and the QueueMonitor restart and then choose to browse queues.	As messages are listed at the moment they are delivered, there are no messages in the MessageMonitor until new deliveries occur.

QueueMonitor Application (PTP)

The QueueMonitor moves through a queue, listing the active messages it finds as it examines the queue.

Starting QueueMonitor

To start QueueMonitor:

1. Open a command line console window, change directory to the QueuePTP\QueueMonitor folder.
2. Type `..\..\SonicMQ QueueMonitor` and press **Enter**.

The **Queue Monitor** browser window opens.

Starting a Talk session without a receiver

To start a Talk session without a receiver:

1. Open a command line console window, change directory to the QueuePTP\Talk folder.
2. Type `..\..\SonicMQ Talk -u Talk1 -qs SampleQ1` and press **Enter**.

The **Talk** session **Talk1**, having no receiver, is started in the **Talk** console window.

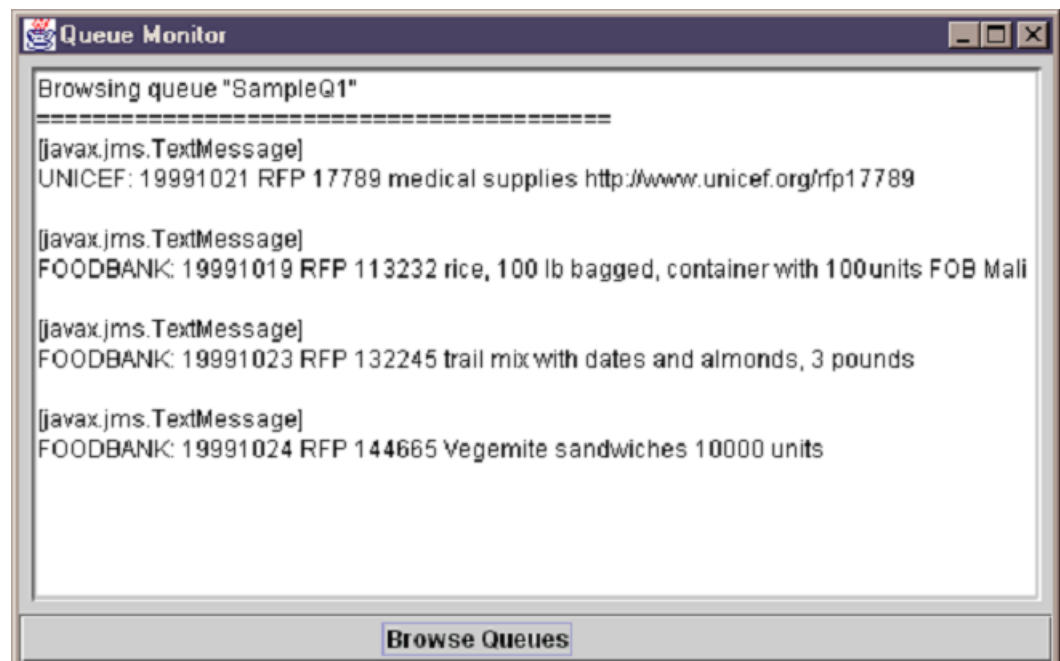
Enqueuing messages and then browse the queue

To enqueue messages and then browse the queue:

1. In the **Talk** window, type some text and then press **Enter**.
Repeat a few times.
2. In the **Queue Monitor** window, click **Browse Queues** to scan the queues and display their contents.

The **Queue Monitor** appears similar to the window shown in the following figure:

Figure 22: Queue Monitor Window



Receiving the queued messages

To receive the queued messages:

The messages that are waiting on the queue will get delivered to the next receiver who chooses to receive from that queue.

warning: If you do not perform this procedure the stored messages will be received in the next application that receives on that queue.

1. In the **Talk** console window, press **Ctrl+C**.
The application stops.
2. Type `..\..\SonicMQ Talk -u FlushQ1 -qr SampleQ1` and press **Enter**.
The enqueued messages are delivered to the queue receiver.

Stopping the sample

To stop the sample:

1. In the console windows, press **Ctrl+C**. The application stops.
2. In the **Queue Monitor** window, click the close button.

MessageMonitor Application (Pub/Sub)

The **MessageMonitor** sample application provides an example of a supervisory application with a graphical interface. By subscribing to all topics in the topic hierarchy, the application listens for any message activity then displays each message in its window.

Starting MessageMonitor

To start MessageMonitor:

Open a command line console window, change directory to the `TopicPubSub\MessageMonitor` folder, then enter: `..\..\SonicMQ MessageMonitor`

The **MessageMonitor** Java window opens.

Running a Chat session to send messages to the MessageMonitor

To run a Chat session to send messages to the MessageMonitor

1. Open a command line console window, change directory to the `TopicPubSub\Chat` folder, then enter: `..\..\SonicMQ Chat -u Chatter`
This command starts a **Chat** session for the user **Chatter**.
2. Type any text in the **Chat** console window, then press **Enter**.

The text is displayed in the **Chat** window and the **MessageMonitor** window. If you send more messages, each one appends to the list displayed, as shown in the following figure:

Figure 23: Message Monitor Window



3. Click the **Clear** button. The list is emptied.

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 as **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. To determine when the transaction is complete, the programmer must:

- 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!**.

Note: If you are interested in exploring global transactions with two-phase commits in a sample, see the sample in [Distributed Transactions Using XA Resources](#) .

TransactedTalk Application (PTP)

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

Starting TransactedTalk sessions

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 command line console window, change directory to the QueuePTP\TransactedTalk folder, then enter: `..\..\SonicMQ TransactedTalk -u Accounting -qr SampleQ1 -qs SampleQ2`

This command starts a **TransactedTalk** session for the user **Accounting**.

2. Open another command line console window, change directory to the QueuePTP\TransactedTalk folder, then enter: `..\..\SonicMQ TransactedTalk -u Operations -qr SampleQ2 -qs SampleQ1`

This command starts a **TransactedTalk** session for the user **Operations**.

Building a PTP transaction and committing it

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.

Building a PTP transaction and rolling it back

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 will form a new transaction. Any messages you resend will be redelivered.

TransactedChat Application (Pub/Sub)

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

Starting Pub/Sub TransactedChat sessions

To start Pub/Sub TransactedChat sessions:

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

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

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

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

Building a Pub/Sub transaction and committing it

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 will form a new transaction.

Building a Pub/Sub transaction and rolling it back

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 will form a new transaction. Any messages you resend will be redelivered.

Reviewing the Transaction Samples

The transaction samples show:

- The transaction scope is between the client in the JMS 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. The following 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 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.
- Transactions are 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. For example, an XML message enables the publisher to send a package of messages and the subscriber to interpret the set of packaged entries as a single message. See [XML Messages](#) on page 64 for details.
- 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.

You can explore this behavior by modifying **TransactedTalk.java** to set the receive session to be transacted, like this:

```
receiveSession =  
connect.createSession(true, javax.jms.Session.AUTO_ACKNOWLEDGE);
```

Then follow the send session commit line and send session rollback line with similar statements for the receive session like this:

```
sendSession.rollback();
receiveSession.rollback();
...
sendSession.commit();
receiveSession.commit();
```

Start the two sessions described in the **TransactedTalk** sample, then run **QueueMonitor** sample. Notice that whether you commit or roll back, no messages stay in the queue. Stop the **TransactedTalk** sessions and the refresh the queue monitor. Note that the messages sent since the last commit were all reinstated in the queue.

For more information, see [Transacted Sessions](#) on page 171.

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 could 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 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 show techniques for monitoring a connection for exceptions and re-establishing the connection if it has been dropped.

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 Management Console runtime.

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

```
javax.jms.IllegalStateException: The session is closed
```

This error occurs because **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 failing 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 ensures 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](#).

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, would not be necessary with a fault-tolerant connection, because the fault-tolerant connection is able to automatically retry 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 set up.

When a fault-tolerant connection encounters a problem and is able to reconnect, your application does not get 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 the exception. Retrying the connection might not make sense if the automatic retry was unsuccessful.

For detailed information about fault-tolerant connections, see [Fault-Tolerant Connections](#).

ReliableTalk Application (PTP)

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

To run the ReliableTalk sample:

1. Open a command line console window, change directory to the `QueuePTP\ReliableTalk` folder, then enter: `..\..\SonicMQ 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, as shown:

```
[ MESSAGE RECEIVED ] AlwaysUp: Hello
There is a problem with the connection.
JMSEException: Connection dropped
Please wait while the application tries to re-establish the connection...
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
```

```
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
```

4. Restart the container and broker by using the Windows **Start** menu command or the **startmf** script. The **ReliableTalk** application reconnects, as shown:

```
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 command line console window, change directory to the `TopicPubSub\ReliableChat` folder, then type: `..\..\SonicMQ 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.

Persistent Storage Application (PTP)

When a message is sent to a queue, the sender can take steps to assure that messages sent are placed on a particular queue by specifying some additional requirements:

- Set the message delivery mode to **PERSISTENT** — The message is logged before the producer is acknowledged and is guaranteed to be retained in the final broker's message store until it is either acknowledged as delivered or expires.
- Set the **JMS_SonicMQ_preserveUndelivered** message property to **true** — If the message is for any reason undelivered, retain it.
- Set the **JMS_SonicMQ_notifyUndelivered** message property to **true** — Send notice to the administrator of the broker that manages the queue.

Every broker provides a dead message queue where messages appropriately flagged are moved when they become expired or undeliverable because a destination on that broker or another remote broker puts message delivery into jeopardy.

In the **DeadMessages** sample application, you first modify two settings in the Management Console that control the broker's periodic checks of queues for expired messages. You then start a session and create a sender to **SampleQ1**. You create a **PERSISTENT** message that has a short time-to-live (so that it will expire). Because this message is **PERSISTENT** and will expire, the message will be sent to the DMQ after it expires. You send the message to **SampleQ1**, then observe the message on queue browsers on **SampleQ1** and the DMQ. Finally, you start the **DeadMessages** sample application that receives messages on the DMQ and displays them in a Java window.

Note: Dynamic routing exposes several other reasons a message could get enqueued in the Dead Message Queue. In a variation of this sample, a message could be unexpired yet become undeliverable because it is sent to a bad node (such as **BadNode::SampleQ1**) or a bad destination (such as **::BadQ**). See the “Guaranteeing Messages” chapter in the *Aurea SonicMQ Application Programming Guide* for detailed examples of each reason code.

Changing queue cleanup settings in the Management Console

To change queue cleanup settings in the Management Console:

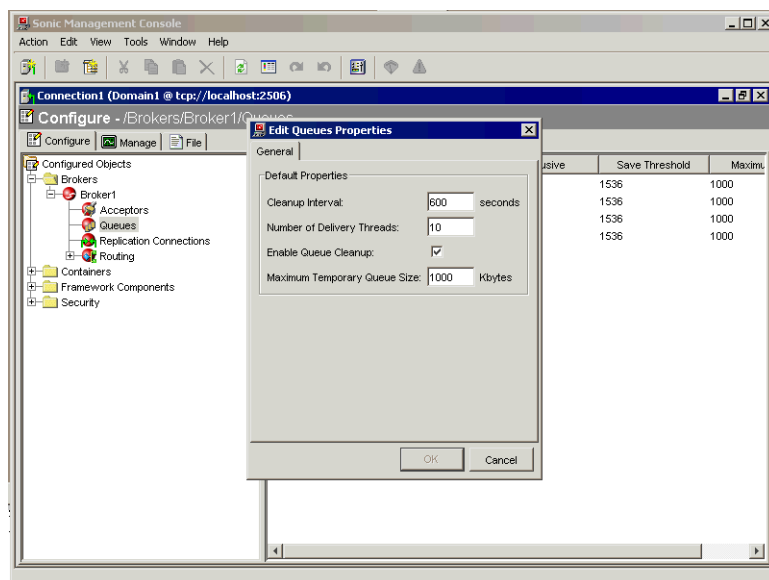
1. Start the SonicMQ container and broker (or confirm that they are already running), then start the Management Console.

See [Starting the SonicMQ® Container and Management Console](#) on page 56 for instructions. See the *Aurea SonicMQ Configuration and Management Guide* for detailed instructions about working with the Management Console.

2. In the Management Console, click the **Configure** tab.
3. In the left panel of the Management Console, expand the node for your broker connection, right-click on the **Queues** node and select **Properties**.

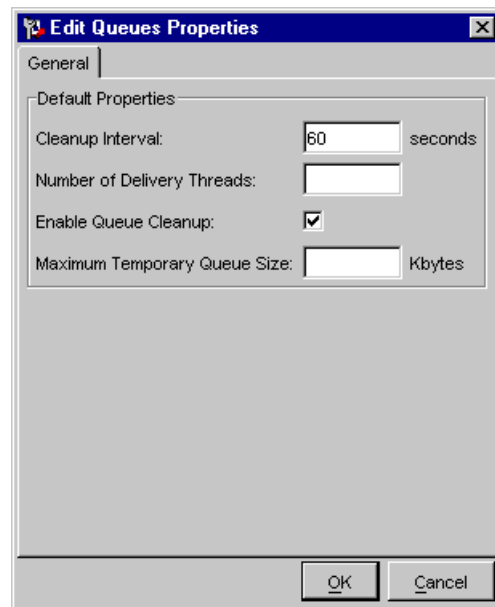
The **Properties** dialog box opens, as shown in the following figure:

Figure 24: Queue Properties in Management Console



4. Set the queue cleanup properties as shown in the following figure:

Figure 25: Set Queue Cleanup Interval



- a) Enter a value in the **Cleanup Interval** field. For this example, enter **60** [seconds].
- b) Make sure the **Enable Cleanup Interval** check box is activated.
- c) Click **OK**.

The cleanup interval is set for the broker.

5. Reload the broker to activate the new cleanup interval:
 - a) In the Management Console, click the **Manage** tab.
 - b) In the left panel, expand the **Containers** node and right-click the node for your broker.
 - c) From the pop-up menu, choose **Operations > Reload**.
 - d) Click **Yes** in the confirmation dialog box that opens.

The broker is reloaded, and the cleanup interval is activated.

Creating a queue session, queue sender, and queue browsers

To create a queue session, queue sender, and queue browsers:

1. Start the JMS Test client.
See [Using the JMS Test Client](#) for instructions.
2. To connect to a broker, click **Message Brokers** in the left panel of the JMS Test client window.
In the right panel:
 - a) In the **Broker Host** field, enter **localhost:2506**.
 - b) In the **Connect ID** field, enter **Conn1**.
 - c) In the **User** field, enter **Administrator**.
 - d) In the **Password** field, enter **Administrator**.
 - e) Click **Connect**.

A node for this connection appears under the **Message Brokers** node in the left panel, and the connection appears in the list of connections in the lower right panel.

3. To create a new queue session, in the left panel click the node for the broker you just connected to: **localhost:2506:Conn1**.

In **Create New Session** area of the right panel:

- a) In the **Name** field, enter **Session1** for the new session.
- b) In the **Type** field, select **Queue** from the pull-down list.
- c) Click **Create**.

A node for the new queue session appears under the node for your broker connection in the left panel. The queue session is listed in the **Established Sessions** area in the right panel.

4. To create a queue sender, in the left panel, click the **Senders** node under **QueueSession:Session1** node.

In the **Create New Sender** area of the right panel:

- a) In the **Queue** field, enter `SampleQ1` as the queue name.
- b) Click **Create**.

A node for **SampleQ1** appears under the **Senders** node in the left panel, and the new sender is listed in the **Established Senders** area in the right panel. This new sender will send messages to **SampleQ1**.

5. Create queue browsers for **SampleQ1** and the DMQ:

- a) In the left panel, click the **Browsers** node.
- b) In the **Create New Browser** area, enter **SampleQ1** in the **Name** field. Click **Create**.

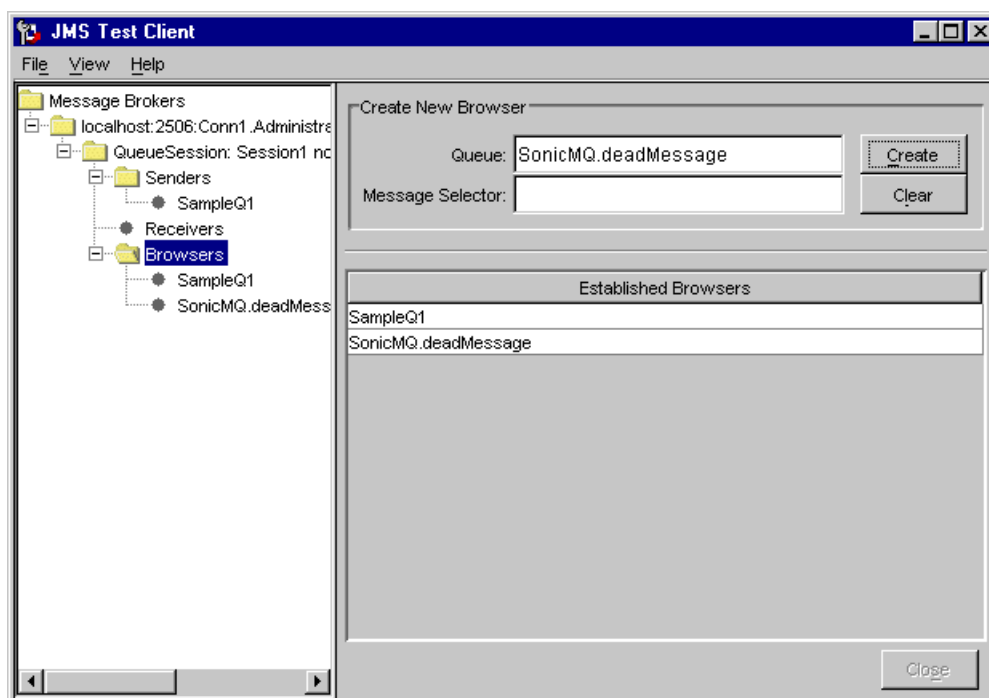
A queue browser is created for **SampleQ1**.

- c) In the **Create New Browser** area, enter **SonicMQ.deadMessage** in the **Name** field, then click **Create**.

A queue browser is created for the DMQ.

The following figure shows the two queue browsers created for this queue session.

Figure 26: Create Queue Browsers



You will use these browsers to watch the message move to the dead message queue after it has expired and the cleanup interval has passed.

Creating and sending a PERSISTENT message that will expire

To create and send a PERSISTENT message that will expire:

1. To create and send a message to **SampleQ1**, in the left panel click the node for your queue sender: **Sender: SampleQ1**.

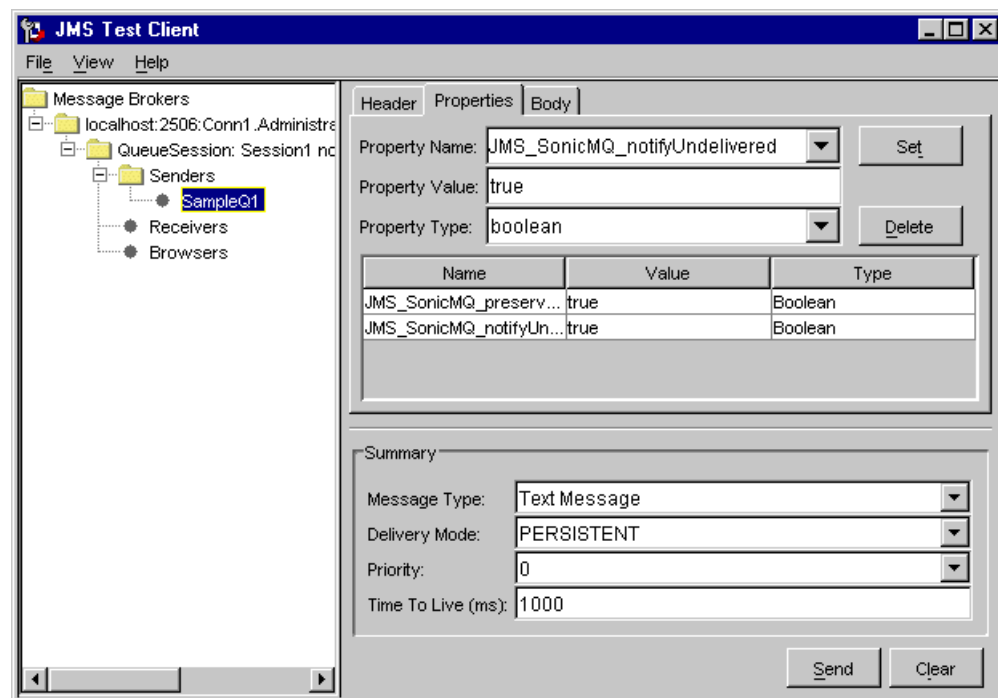
Under the **Body** tab in the right panel:

- Enter some text for the body of the message.
- In the **Summary** area, choose the **Delivery Mode** option **PERSISTENT** and enter a **Time To Live** value greater than zero, for example, **30000 [ms]**.

Under the **Properties** tab in the right panel:

- Choose the **Property Name** `JMS_SonicMQ_preserveUndelivered` from the pull-down list. In the **Property Value** field, enter `true`. Click **Set**.
- Choose the **Property Name** `JMS_SonicMQ_notifyUndelivered` from the pull-down list. Click **Set**. The **Property Value** `true` is carried forward, as shown in the following figure:

Figure 27: Persistent Message



2. Click **Send** to send the message to **SampleQ1**.

The message will be enqueued on **SampleQ1** for 30 seconds, the **Time To Live** value that you specified. If you had put an active receiver on that queue before the message expired, you would see that the message was listed in **SampleQ1**, awaiting receivers on that queue. Then your receiver would have taken it off the queue. However, the purpose of this sample is to demonstrate a message that expires while waiting for a receiver. For that reason, you created queue browsers that allow you to browse the messages without removing them from the queue.

Messages that have expired are not removed from the original queue until they are examined by the broker and found to be expired.

Viewing messages on SampleQ1 and the DMQ

To view messages on SampleQ1 and the DMQ:

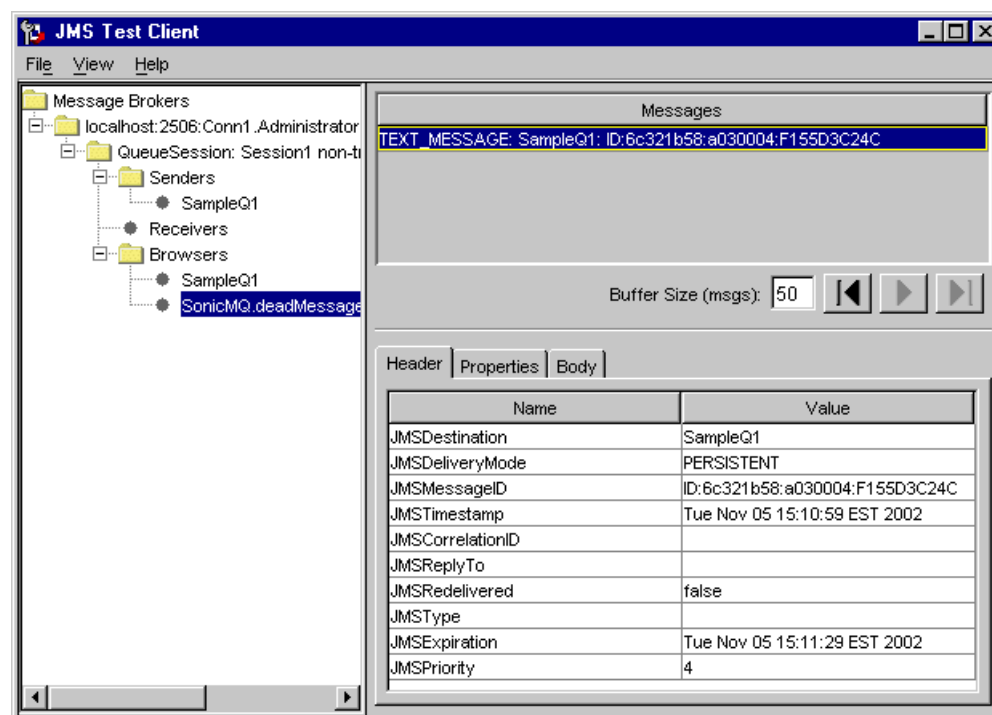
1. In the left panel of the JMS Test client, click the **SampleQ1** browser node, then click the left arrow button in the right panel.

If it has been less than a minute (the time you set for the queue cleanup interval) since the message expired, the message is listed in the **Messages** area.

2. Click the **SonicMQ.deadMessages** browser node, then click the left arrow in the right panel.

When more than a minute has passed since the message expired, the message will appear on the DMQ and you will see it in the browser, as shown in the following figure:

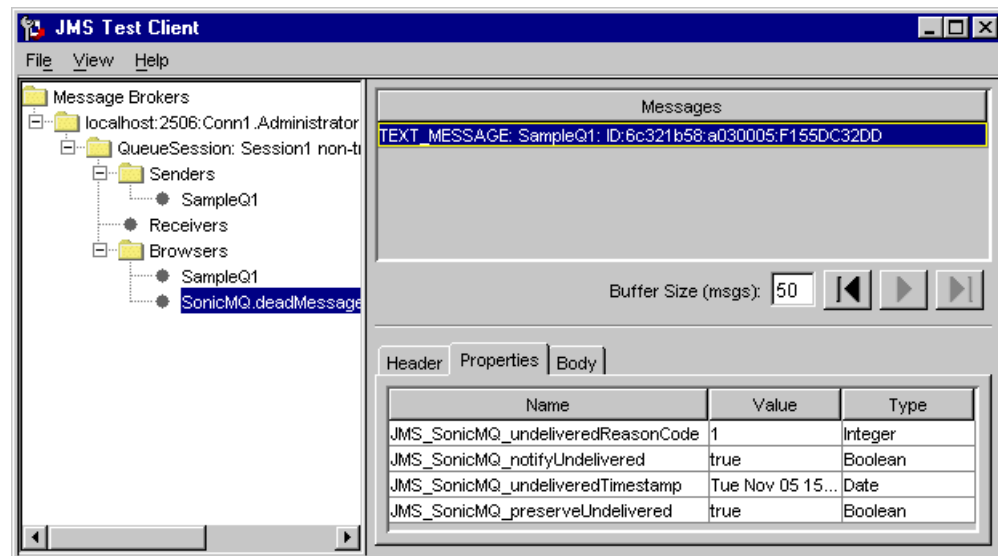
Figure 28: Expired PERSISTENT Message on DMQ



3. Click the **Properties** tab.

The following figure shows the properties of the undelivered, expired message.

Figure 29: Expired PERSISTENT Message Properties

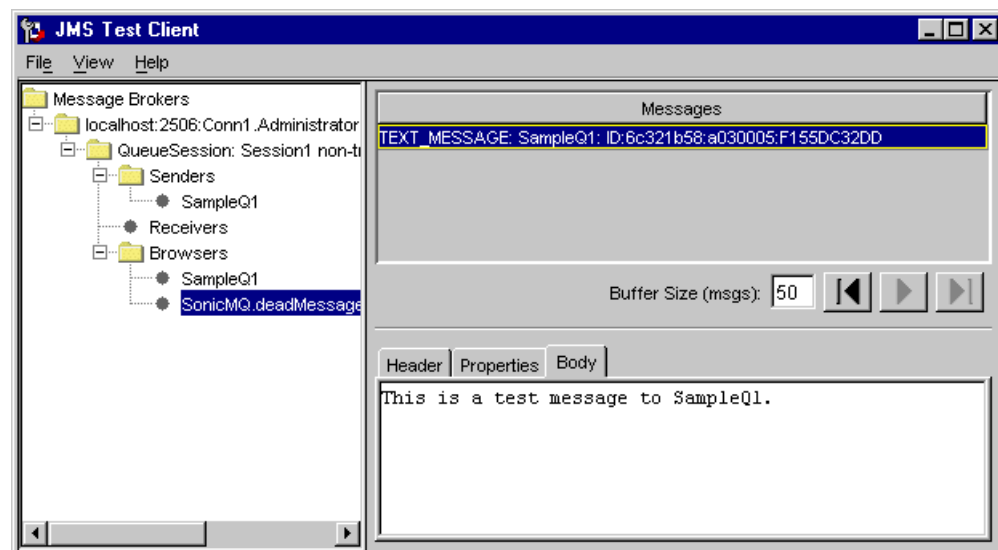


The properties include the original settings to preserve and notify when undelivered. The undelivered timestamp indicates the time of dequeuing into the DMQ. The reason code, 1, indicates that the message expired.

- Click the **Body** tab.

The body is unchanged, as shown in the following figure:

Figure 30: Expired PERSISTENT Message Body



- Click the left arrow in the **SampleQ1** browser to see that the message has been removed from that queue.

Expired messages are examined and, with the appropriate properties set, are transferred to the dead message queue. The property you set instructs the broker to transfer the expired message to the DMQ, placing it under administrative control with no expiration. The message must now be explicitly flushed or dequeued. You can remove this message from the DMQ by creating a receiver to that queue, or by running an application that takes a message off the DMQ. The following procedure explains how to run the Dead Message browser sample application to remove the message from the DMQ and display it in a Java window.

Running the DeadMessages browser sample

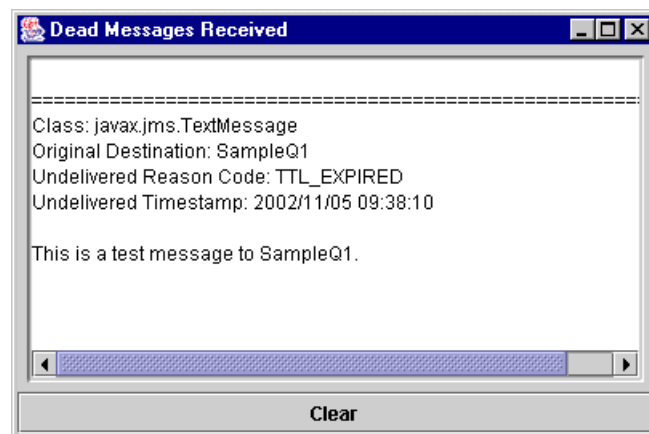
To run the **DeadMessages** browser sample:

1. Open a command line console window, change directory to the `QueuePTP\DeadMessages` folder.
2. Type `..\..\SonicMQ DeadMessages` and press **Enter**.

The **Dead Messages Received** Java window opens.

3. The dead messages are listed in the **Dead Messages Received** window, as shown in the following figure:

Figure 31: Dead Messages Received Browser



4. In the JMS Test client, click the **SonicMQ.deadMessages** browser node, then click the left arrow in the right panel.

The message has been removed from the DMQ by the **Dead Message** browser sample application.

A management application might clone the body into a new message and use some business logic to reroute the message to an optional or fallback destination.

While expiration is common to all messaging deployments, there are several other reasons a messages could be in-doubt or undeliverable in a dynamic routing architecture.

See the *Aurea SonicMQ Application Programming Guide* for information about using the dead message queue and the dynamic routing architecture.

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 that were 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.

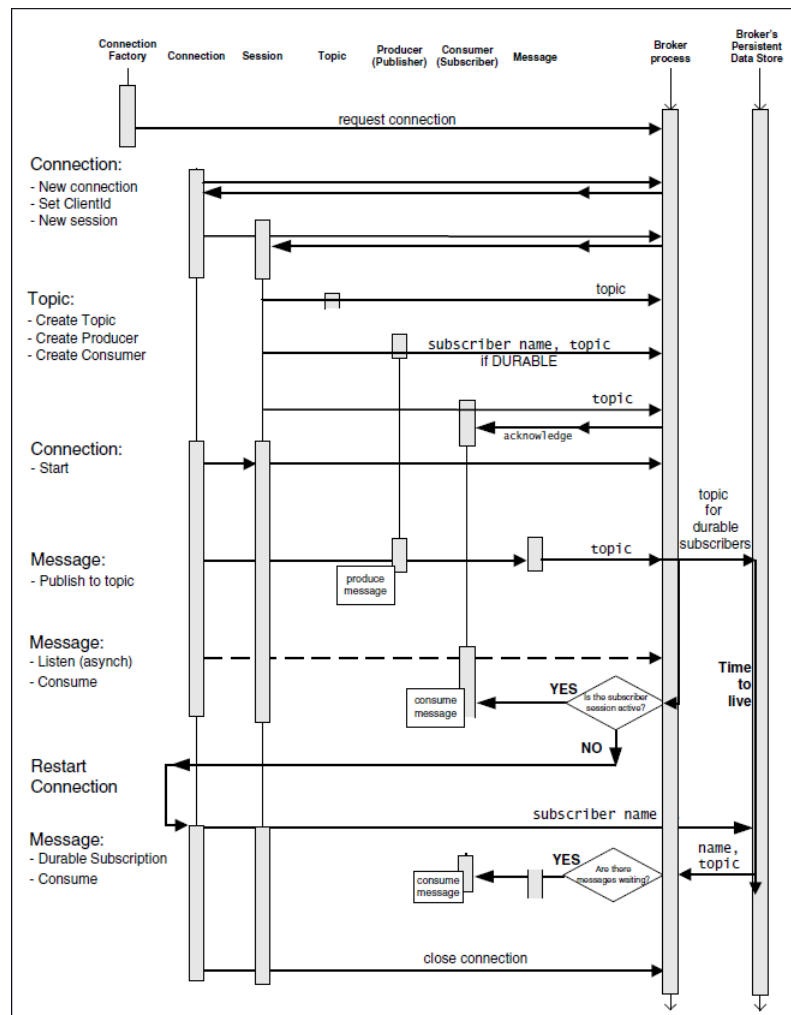
In an application, there are only a few changes to set up a subscriber as a durable subscriber. Where **Chat** was coded as:

```
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.

The following figure shows what occurs when the subscriber requests an extra effort to ensure delivery.

Figure 32: Sequence Diagram for the DurableChat Application



To start DurableChat sessions:

1. Open a command line console window, change directory to TopicPubSub\DurableChat folder, then enter: `..\..\SonicMQ DurableChat -u AlwaysUp`

This command starts a **DurableChat** session for the user **AlwaysUp**.

2. Open another command line console window, change directory to the `TopicPubSub\DurableChat` folder, then enter: `..\..\SonicMQ 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's consoles.

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

The text is displayed on both subscriber's 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.

While durable, the messages were not implicitly everlasting. 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.

Continuous Producer Demonstrating Client Persistence

Note: This sample requires the ClientPlus libraries for the SonicMQ client. When you have the ClientPlus edition or the Enterprise Plus edition, these features are available to you.

While the **ReliableTalk** sample (see [ReliableChat Application \(Pub/Sub\)](#) on page 79) showed that the client can reconnect when the broker is again available, other features enable the client to continue its work when it is sending messages and the broker connections fails. The SonicMQ ClientPlus has an extended capability that enables the client to establish a message cache on the client where a definable volume of sent messages can be buffered while a connection is re-established. When the connection and session are again active, the oldest messages buffered are sent normally and more recent messages sent continue to accrue in the buffer. When the local store is empty, the use of the local store is transparent.

The applications in the **LocalStore** sample provide the extended feature of client persistence, a way for client application to continue sending messages despite losing connection with the broker. Messages sent by the client are buffered in a persistent store on the client system until connection is established at which time the accrued messages are sent. This section includes two sets of samples, one for each messaging domain. Each set runs a continuous producer that sends and displays a sequence number and a consumer that receives the messages sent. The broker is stopped to effect the local store of produced message. When the broker restarts, the messages are sent and the receiver displays them.

Local Store Sample (PTP)

This example includes two applications to continuously send and receive messages using the PTP messaging model.

To send Chat messages even when the broker connection stops:

1. Open a command line console window, change directory to the `ClientPlus\LocalStore` folder, then enter: `..\..\SonicMQ MessageReceiver -u Warehouse -qr SampleQ1`

This command starts a **MessageReceiver** session for the receiver **Warehouse**.

2. Open another command line console window, change directory to the `ClientPlus\LocalStore` folder, then type: `..\..\SonicMQ ContinuousSender -u HandHeld -qs SampleQ1`

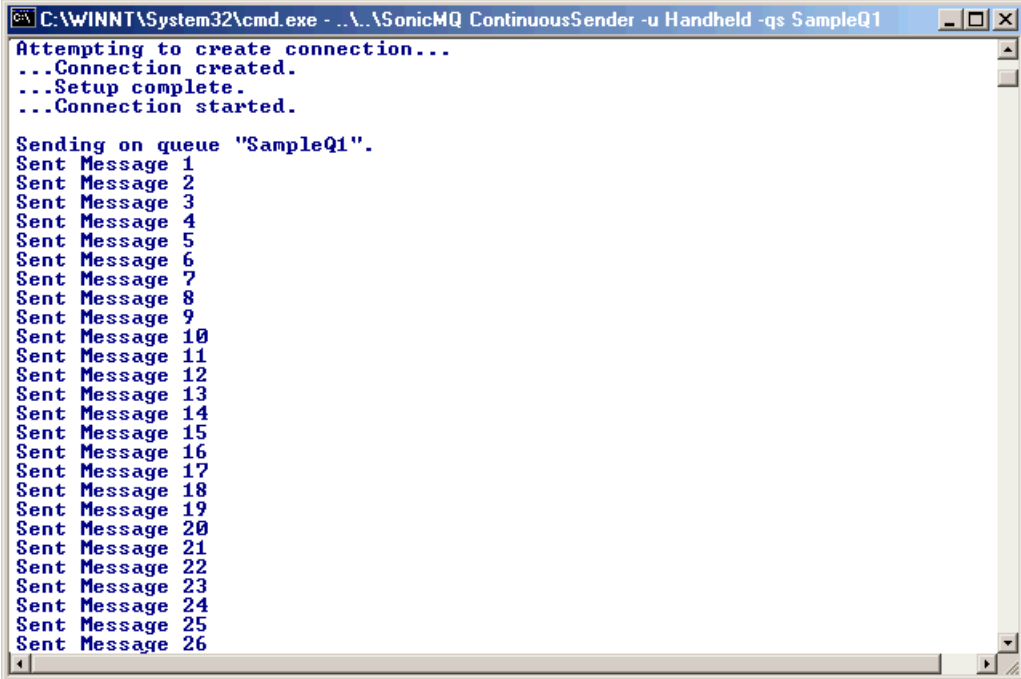
This command starts a **MessageReceiver** session for the sender **HandHeld**.

The sender connects and starts sending messages to the queue. The receiver takes the enqueued messages from the queue.

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

The connection is broken for both the sender and the receiver. The receiver tries repeatedly to reconnect. The sender continues to send messages without pause while the broker is unavailable, as shown in the following figure:

Figure 33: ContinuousSender Sample Sends Without Pause



```

C:\WINNT\System32\cmd.exe - ..\..\SonicMQ ContinuousSender -u Handheld -qs SampleQ1
Attempting to create connection...
...Connection created.
...Setup complete.
...Connection started.

Sending on queue "SampleQ1".
Sent Message 1
Sent Message 2
Sent Message 3
Sent Message 4
Sent Message 5
Sent Message 6
Sent Message 7
Sent Message 8
Sent Message 9
Sent Message 10
Sent Message 11
Sent Message 12
Sent Message 13
Sent Message 14
Sent Message 15
Sent Message 16
Sent Message 17
Sent Message 18
Sent Message 19
Sent Message 20
Sent Message 21
Sent Message 22
Sent Message 23
Sent Message 24
Sent Message 25
Sent Message 26

```

4. Restart the broker by using its Windows **Start** menu command or the **startmf** script. Both applications reconnect to the broker.

After reconnecting, the **MessageReceiver** application gets all the sent messages from its local store, including those sent while the broker connection was broken, as shown in the following figure:

Figure 34: MessageReceiver Sample Handling Disconnection

```

C:\WINNT\System32\cmd.exe - ..\..\SonicMQ MessageReceiver -u Warehouse -qr SampleQ1
Received Message 14
Received Message 15
Received Message 16

There is a problem with the connection.
JMSEException: Connection dropped
Please wait while the application tries to re-establish the connection...
Attempting to create connection...

There is a problem with the connection.
JMSEException: An open connection has not been established
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting to create connection...
...Connection created.
...Setup complete.
Received Message 17
...Connection started.

Receiving messages on queue "SampleQ1".

Press CTRL-C to exit.
Received Message 18
Received Message 19
Received Message 20
Received Message 21
Received Message 22

```

You can stop both sessions by pressing **Ctrl+C** in the sender and receiver console windows before proceeding to the next sample.

Local Store Sample (Pub/Sub)

This example includes two applications to continuously publish and subscribe using the Pub/Sub messaging model.

To run the LocalStore Pub/Sub sample:

1. Open a command line console window, change directory to the `ClientPlus\LocalStore` folder, then enter: `..\..\SonicMQ ContinuousPublisher -u Wireless`
The publisher connects and starts publishing messages to the **LocalStore.sample** topic.
2. Open another command line console window, change directory to the `ClientPlus\LocalStore` folder, then enter: `..\..\SonicMQ MessageSubscriber -u Customer`
The subscriber connects and receives messages published to the **LocalStore.sample** topic.
3. Stop the broker by pressing **Ctrl+C** in the broker window.
The connection is broken for both the publisher and the subscriber. The subscriber tries repeatedly to reconnect.
4. Restart the container and broker using Windows **Start** menu command or the **startmf** script.
Both applications reconnect to the broker.

After reconnecting, the **MessageSubscriber** application gets all the published messages from its local store, including those published while the broker connection was broken.

You can stop both sessions by pressing **Ctrl+C** in the sender and receiver console windows before proceeding to the next sample.

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 so that you create a reliable, persistent talk and a reliable, durable chat. The source code of these samples is readily transferable into your applications.

The ClientPlus feature of persistence on the client shows how clients can store messages to provide a higher level of reliability to supporting applications that need to produce messages at will. There are also other facets to consider for optimal QoS, including the various security, encryption, access control, and transport protocols. See the *Aurea 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 get 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: standardize the case of the text sent—receive text and send back the same text as either all uppercase characters or all lowercase characters—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: **JMSReplyTo** and **JMSCorrelationID** are used as a suggested design pattern established as a part of the JMS specification. The application programmer ultimately decides how these fields are used, if they are used at all.

The sample applications use JMS sample 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: `..\..\SonicMQ Requestor -u QRequestor`
This command starts a PTP **Requestor** session for the user **QRequestor**.
3. In the other console window enter: `..\..\SonicMQ 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**.

Testing a PTP request and reply

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 gets 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 command line console windows. In each shell change directory to the **TopicPubSub\RequestReply** folder.
2. In one of the windows enter: `..\..\SonicMQ 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: `..\..\SonicMQ Requestor`
This command starts a Pub/Sub **Requestor** session.

Testing a Pub/Sub request and reply

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 gets 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 will be, at most, one PTP reply.
- Using message header fields (**JMSReplyTo** and **JMSCorrelationID**) and the requestor sample classes (**javax.jms.TopicRequestor** and **javax.jms.QueueRequestor**) are suggested implementations for request-and-reply behavior in JMS. These examples, however, require you to use model-specific JMS Version 1.02b interfaces. For a description about how you can accomplish request-reply functionality using interfaces common to both models, see [Reply-to Mechanisms](#).

Selection, Group, and Wild Card Samples

While specific queues and topics provide focused content nodes for messages that are of interest to application producers and consumers, there are circumstances where the programmer might want to control what subsets of messages a receiver actively selects, or what subsets of messages a queue receiver is passively assigned to accept. The two techniques are mutually exclusive for queue receivers. Either:

- The receiver decides which messages it wants through message selection based on syntax much like an SQL **WHERE** clause.
- The receiver is assigned to one or more message groups defined by the queue sender and dispatched by the broker.

A variation of selection is also explored in this section. SonicMQ lets you use dot-delimited naming hierarchies so that a topic consumer can create wildcards that express interest in receiving messages in leaf topic levels without knowing specific topic names.

Message Selection: SelectorTalk and SelectorChat

While a consumer could declare each destination of interest, the dynamic naming of topics (assuming there are no security constraints) means that a subscriber application might need to scan many topics.

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 will be consumed. When the subscriber message traffic is a burden and server resources can handle it, you can command a Pub/Sub message selector to filter the messages on the server by calling `factory.setSelectorAtBroker(true)` on the **ConnectionFactory**.

SelectorTalk Application (PTP)

The **SelectorTalk sample** application starts by declaring a selector **String-value** that will 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 command line console window, change directory to the `QueuePTP\SelectorTalk` folder, then enter: `..\..\SonicMQ 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 command line console window, change directory to the `QueuePTP\SelectorTalk` folder, then enter: `..\..\SonicMQ 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: `..\..\SonicMQ 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 will be 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 command line console window, change directory to the `TopicPubSub\SelectorChat` folder, then enter: `..\..\SonicMQ SelectorChat -u Closer -s Sales`

This command starts a **SelectorChat** session for the user **Closer** with the selector string **Sales**.

2. Open another command line console window, change directory to the `TopicPubSub\SelectorChat` folder, then enter: `..\..\SonicMQ 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: `..\..\SonicMQ 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.

MessageGroupTalk (PTP)

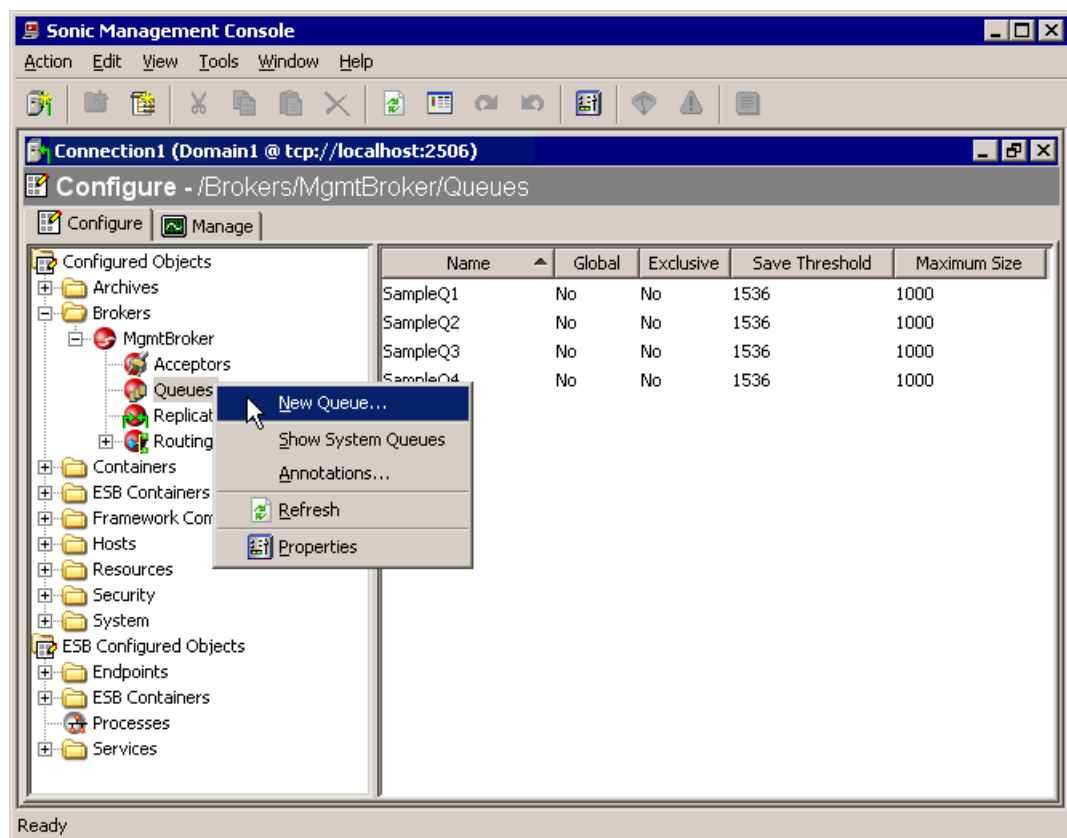
In the **MessageGroupTalk** sample, a queue is set up on the broker that will enable message grouping. Then producer applications send messages with message group identifiers. As consumer applications are allocated messages by the broker, they are bound to message groups.

To set up a queue that enables message grouping:

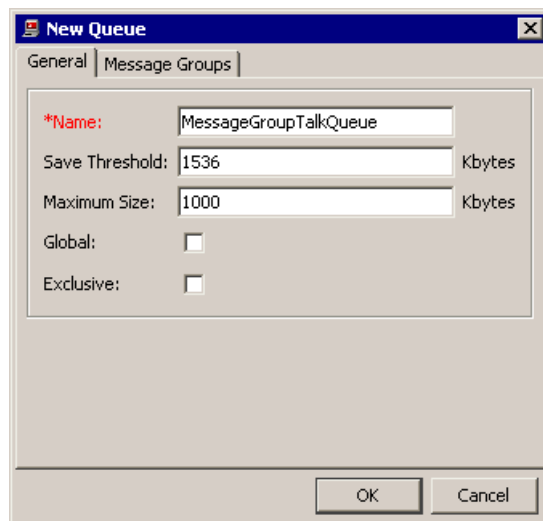
1. Start the SonicMQ container and broker (or confirm that they are already running), then start the Management Console.

See [Starting the SonicMQ® Container and Management Console](#) on page 56 for instructions.

2. In the Management Console, click the **Configure** tab.
3. In the left panel of the Management Console, expand the node for your broker connection, right-click on the **Queues** node and select **New Queue**.



4. In the **New Queue** dialog box, enter the queue name **MessageGroupTalkQueue**, as shown:



5. Click the **Message Groups** tab, and then select (check) **Enable Message Group Handling**, as shown:



6. Click **OK**.

The queue is created and ready to dispatch messages that request grouping on dispatch to consumers.

Running MessageGroupTalk sessions

To run **MessageGroupTalk** sessions:

1. Open a command line console window, change directory to the `QueuePTP\MessageGroupTalk` folder, then enter: `..\..\SonicMQ MessageGroupTalk -u Claims1 -qr MessageGroupTalkQueue`

This command starts a **MessageGroupTalk** receiver session for the user **Claims1**.

2. Open another console window, then enter: `..\..\SonicMQ MessageGroupTalk -u Claims2 -qr MessageGroupTalkQueue`

This command also starts a **MessageGroupTalk** receiver session for the user **Claims2**.

These are undifferentiated receivers. Without message grouping, they would take turns receiving the messages off the queue.

3. Open a console window, then enter: `..\..\SonicMQ MessageGroupTalk -u Adjuster1 -qs MessageGroupTalkQueue -g ABC`

This command starts a **MessageGroupTalk** send session for the group **ABC**.

4. Open another console window, then enter: `..\..\SonicMQ MessageGroupTalk -u Adjuster2 -qs MessageGroupTalkQueue -g DEF`

This command starts a **MessageGroupTalk** send session for the group **DEF**.

5. In the **Adjuster1** window, type any text and then press **Enter**.

The text is displayed in the window of its assigned receiver.

Notice that the message indicates the message group. That message group is bound to that receiver as long as the broker and the receiver are running. (There is a settable timeout based on inactivity, and the sender can explicitly tell the broker to close the assigned receiver. See [Using Message Grouping](#).)

6. Enter more messages in the **Adjuster1** window.

The text is also displayed in the window of the assigned receiver to group **ABC**.

7. In the **Adjuster2** window, type any text and then press **Enter**.

The text is displayed in the window of its assigned receiver, which likely is the other receiver.

8. Enter more messages in the **Adjuster2** window.

The text is displayed in the window of the assigned **Claims** receiver to group **DEF**.

Keep these windows open while you explore other message grouping behaviors.

Reassigning a message group's receiver

To reassign a message group's receiver:

1. Stop the **Claims1** receiver session by pressing **Ctrl+C**. Then start it again using the same commandline as before.
2. In each of the sender (**Adjuster1** and **Adjuster2**) windows, enter text. The messages are both received in the **Claims2** window. When you closed the other receiver, its message group was reassigned by the broker to another active receiver.

Adding another sender (and group) and another receiver

To add another sender (and group) and another receiver:

1. Open another console window, then enter: `..\..\SonicMQ MessageGroupTalk -u Claims3 -qr MessageGroupTalkQueue`

This command starts a **MessageGroupTalk** receiver session for the user **Claims1**.

2. Open a console window, then enter: `..\..\SonicMQ MessageGroupTalk -u Adjuster3 -qs MessageGroupTalkQueue -g GHI`
3. In the **Adjuster3** window, enter a few messages.

Either the new receiver or the restarted receiver get the new group.

You can continue to explore the behaviors with other exercises:

- Stop **Claims2** and then restart it. Send messages from each **Adjuster**. Note the two groups assigned to that receiver are assigned to the other receivers.
- Stop an **Adjuster**, change its group name to the same group name as an active **Adjuster**, and then restart it. Send messages from each **Adjuster**. Note that senders to the same group are all received by the same assigned group receiver. Also notice that starting and stopping the sender had no impact on the assigned group receivers.

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 command line console window, change directory to the `TopicPubSub\HierarchicalChat` folder then enter: `..\..\SonicMQ 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 command line console window, change directory to the `TopicPubSub\HierarchicalChat` folder then enter: `..\..\SonicMQ 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**

Running HierarchicalChat

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, Group, and Wild Card Samples

While selector strings can provide a variety of ways to qualify what messages will be chosen for receipt by a consumer, the overhead in the evaluation of the selectors can slow down overall system performance. See [Message Selection](#) for more information about message selectors.

Message groups enable queue senders in concert with broker queue administrators to focus receivers on the queue on a series of messages that should be consumed in order. While not as strict as exclusive receivers or transactions, message grouping provides pretty good handling of sets messages as identified entirely by the message producers. See [Using Message Grouping](#) for more information about message grouping.

HierarchicalChat illustrates a feature of SonicMQ that can provide 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. See [Hierarchical Name Spaces](#) on page 345 for more information about hierarchical name spaces selectors. For information on hierarchical security, see the *Aurea SonicMQ Deployment Guide*.

Test Loop Sample

A simple loop test lets you experiment with messaging performance.

QueueRoundTrip Application (PTP)

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.

To run QueueRoundTrip:

1. Open a command line console window, change directory to the `QueuePTP\QueueRoundTrip` folder then enter: `..\..\SonicMQ 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, as shown:

```
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: `..\..\SonicMQ 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, as shown:

```
Sending Messages to Temporary Queue... Time for 1000 sends and receives: 5538ms  
Average Time per message: 5.538ms Press enter to continue...
```

Note: This sample lets you evaluate features and is not intended as a performance tool. For information on performance, see the *Aurea SonicMQ Performance Tuning Guide*.

Enhancing the Basic Samples

After exploring the basic samples you can modify the sample source files to learn more about SonicMQ. You need a Java compiler to compile your changes.

Use Common Topics Across Clients

When you run the Pub/Sub samples you might notice that while all the **Chat** applications get Chat messages and all the **DurableChat** applications get **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.java** 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.java** file.
4. Run the new **DurableChat.class** file.

Now 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 will 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

The **RoundTrip** sample application lets 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: This sample lets you evaluate features and is not intended as a performance tool. For information on performance, see the *Aurea 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 SonicMQ sample file `QueuePTP\QueueRoundTrip.java` to establish any of the following behavior changes:
 - Change the **javax.jms.message.DeliveryMode** from **NON_PERSISTENT** to **PERSISTENT**. Run it, then change it to **NON_PERSISTENT_ASYNC**.
 - You could 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 **.java** file.

3. Open a command line console window, change directory to the `QueuePTP\QueueRoundTrip` folder then enter `..\..\SonicMQ QueueRoundTrip -n 100`.
4. Look at the results and compare them to other round trips (see [QueueRoundTrip Application \(PTP\)](#) on page 100).

Modifying the MapMessage to Use Other Data Types

The concept of the **MapMessage** sample application is limited when 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 Java 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 SonicMQ sample file **MapTalk.java** at the lines:

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

2. Add the lines for the **set()** methods (or your 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 will display in the console.

Change this:

```
String content = mapMessage.getString("content");
```

```
System.out.println(sender + ": " + content);
```

to:

```
SString content =
    ("Content: " + mapMessage.getString("content") + "\n" +
     "Distribution: " + mapMessage.getString("Distribution") + "\n" +
     "FiscalYearEnd: " + mapMessage.getString("FiscalYearEnd") + "\n" +
     "LineOfCredit: " + mapMessage.getString("LineOfCredit") + "\n");
System.out.println("MapMessage from " + sender + "\n----- \n" + content);
```

4. Save and compile the edited **.java** file.
5. Run the edited **.class** file.

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

Modifying the XMLMessage Sample to Show More Data

The **XMLDOMTalk** and **XMLDOMChat** samples for the **XMLMessage** type are limited to the data that is input as text as a single content node. While the data collection/validation loops and the data transfers from application data stores are reserved as more advanced exercises, this example demonstrates how well-formed XML data is transformed into **DocNodes** from the **org.w3c.dom.Node** standards.

To extend the XMLDOMChat sample to show more data:

1. Edit the SonicMQ sample file **XMLDOMChat.java** starting after:

```
// Note that the XMLMessage is a Aurea Software extension
progress.message.jclient.XMLMessage xMsg =
...
StringBuffer msg = new StringBuffer();
msg.append("<?xml version=\"1.0\"?>\n");
msg.append("<message>\n");
msg.append("  <sender> + username + "</sender>\n");
msg.append("  <content> + s + "</content>\n");
```

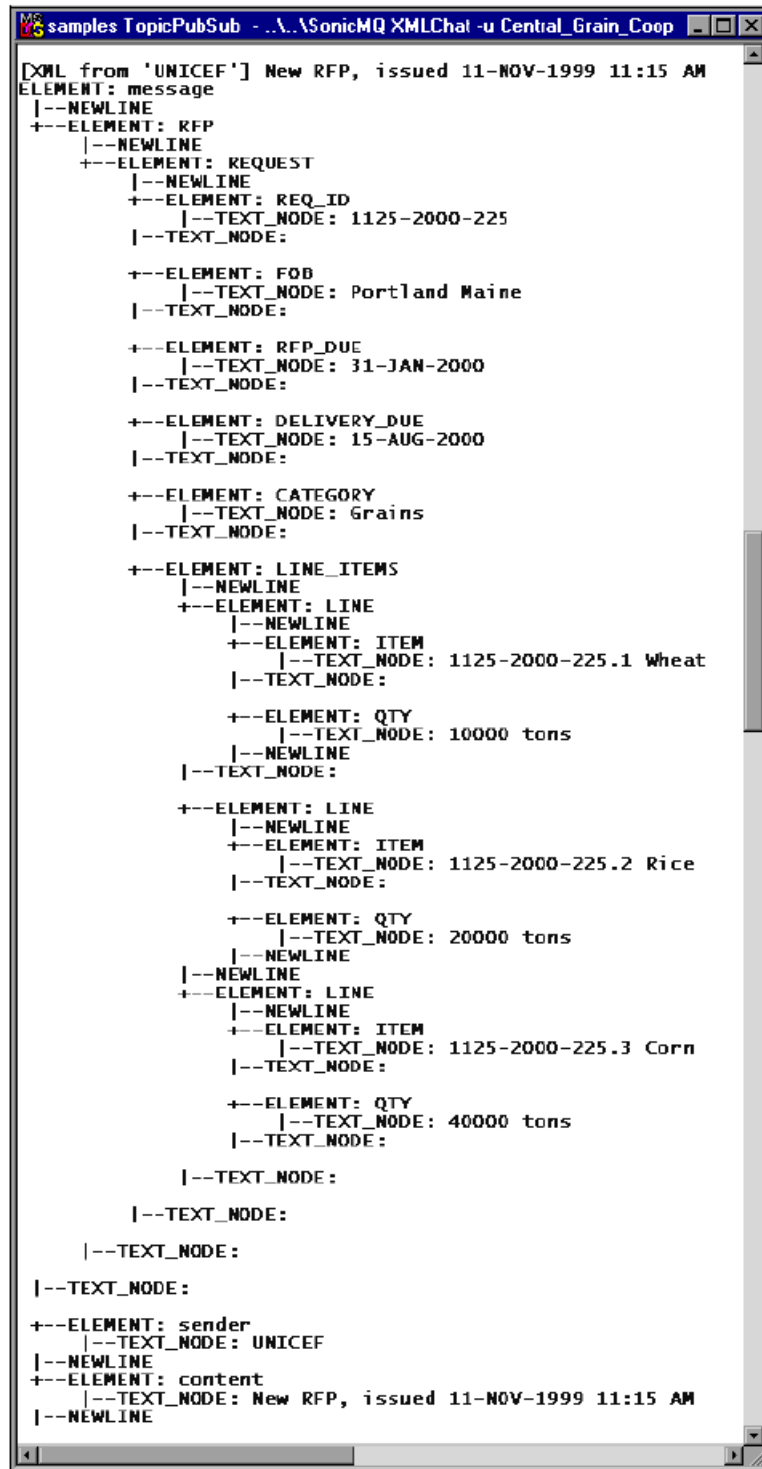
2. Insert the formatted, tagged XML lines you want to append to the message. For example:

```
msg.append("<RFP>\n");
msg.append("<REQUEST>\n");
msg.append("<REQ_ID>1125-2000-225</REQ_ID> \n");
msg.append("<FOB>Portland Maine</FOB> \n");
msg.append("<RFP_DUE>31-JAN-2000</RFP_DUE> \n");
msg.append("<DELIVERY_DUE>15-AUG-2000</DELIVERY_DUE> \n");
msg.append("<CATEGORY>Grains</CATEGORY> \n");
msg.append("<LINE_ITEMS>\n");
msg.append("<LINE>\n");
msg.append("<ITEM>1125-2000-225.1 Wheat</ITEM> \n");
msg.append("<QTY>10000 tons</QTY>\n");
msg.append("</LINE> \n");
msg.append("<LINE>\n");
msg.append("<ITEM>1125-2000-225.2 Rice</ITEM> \n");
msg.append("<QTY>20000 tons</QTY>\n");
msg.append("</LINE>\n");
msg.append("<LINE>\n");
msg.append("<ITEM>1125-2000-225.3 Corn</ITEM> \n");
msg.append("<QTY>40000 tons</QTY> \n");
msg.append("</LINE> \n");
msg.append("</LINE_ITEMS> \n");
msg.append("</REQUEST> \n");
msg.append("</RFP> \n");
msg.append("</message> \n");
```

3. Save and compile the edited .java file.
4. Run the edited .class file.

When you run the application and enter a basic text message, the complete document object model (DOM) is also displayed, similar to the subscriber session listing in the following figure:

Figure 35: XMLMessage Parsed into a Document Object Model



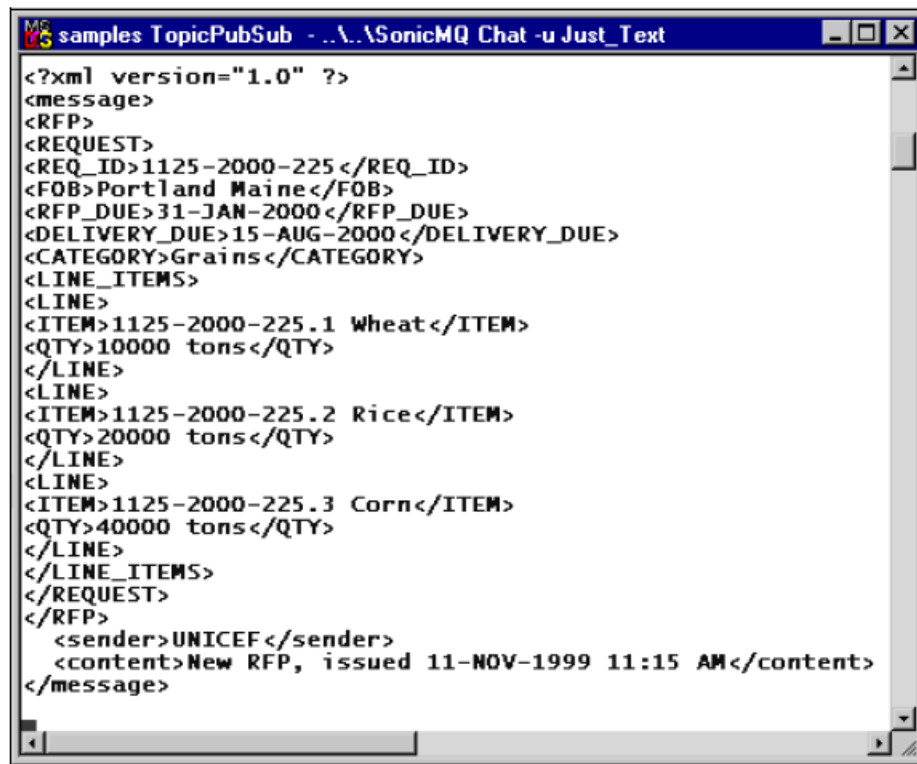
```

samples TopicPubSub - ..\..\SonicMQ XMLChat -u Central_Grain_Coop
[XML from 'UNICEF'] New RFP, issued 11-NOV-1999 11:15 AM
ELEMENT: message
|--NEWLINE
+--ELEMENT: RFP
|--NEWLINE
+--ELEMENT: REQUEST
|--NEWLINE
+--ELEMENT: REQ_ID
|--TEXT_NODE: 1125-2000-225
|--TEXT_NODE:
+--ELEMENT: FOB
|--TEXT_NODE: Portland Maine
|--TEXT_NODE:
+--ELEMENT: RFP_DUE
|--TEXT_NODE: 31-JAN-2000
|--TEXT_NODE:
+--ELEMENT: DELIVERY_DUE
|--TEXT_NODE: 15-AUG-2000
|--TEXT_NODE:
+--ELEMENT: CATEGORY
|--TEXT_NODE: Grains
|--TEXT_NODE:
+--ELEMENT: LINE_ITEMS
|--NEWLINE
+--ELEMENT: LINE
|--NEWLINE
+--ELEMENT: ITEM
|--TEXT_NODE: 1125-2000-225.1 Wheat
|--TEXT_NODE:
+--ELEMENT: QTY
|--TEXT_NODE: 10000 tons
|--NEWLINE
|--TEXT_NODE:
+--ELEMENT: LINE
|--NEWLINE
+--ELEMENT: ITEM
|--TEXT_NODE: 1125-2000-225.2 Rice
|--TEXT_NODE:
+--ELEMENT: QTY
|--TEXT_NODE: 20000 tons
|--NEWLINE
|--TEXT_NODE:
+--ELEMENT: LINE
|--NEWLINE
+--ELEMENT: ITEM
|--TEXT_NODE: 1125-2000-225.3 Corn
|--TEXT_NODE:
+--ELEMENT: QTY
|--TEXT_NODE: 40000 tons
|--TEXT_NODE:
|--TEXT_NODE:
|--TEXT_NODE:
|--TEXT_NODE:
+--ELEMENT: sender
|--TEXT_NODE: UNICEF
|--NEWLINE
+--ELEMENT: content
|--TEXT_NODE: New RFP, issued 11-NOV-1999 11:15 AM
|--NEWLINE

```


Because the data is interpreted in the DOM format only when the message is an instance of an **XMLMessage**, a **Chat** session displays the same message as a **TextMessage**— the XML-tagged text without DOM interpretation, as shown in the following figure:

Figure 36: XMLMessage as Tagged Text



Note: You could have appended the XML tagged lines without the `\n`, suppressing the blank **TEXT_MODE** lines in the DOM. It would, however, make one unbroken text line for general text or raw XML review.

You can continue working with the samples by changing broker settings to explore connection protocols and protocol handlers. You can also enable security on the broker persistent storage mechanism then examine the protocols that provide connection security. For information about using protocols and security, see the *Aurea SonicMQ Configuration and Management Guide*.

SonicMQ® Connections

This chapter explains the programming concepts and actions required to establish and maintain SonicMQ® connections.

For details, see the following topics:

- [Overview of SonicMQ Connections](#)
- [Protocols](#)
- [JVM Command Options](#)
- [Connection Factories and Connections](#)
- [Connecting to SonicMQ Directly](#)
- [Client Persistence](#)
- [Asynchronous Message Delivery](#)
- [Fault-Tolerant Connections](#)
- [Starting, Stopping, and Closing Connections](#)
- [Using Multiple Connections](#)
- [Communication Layer](#)

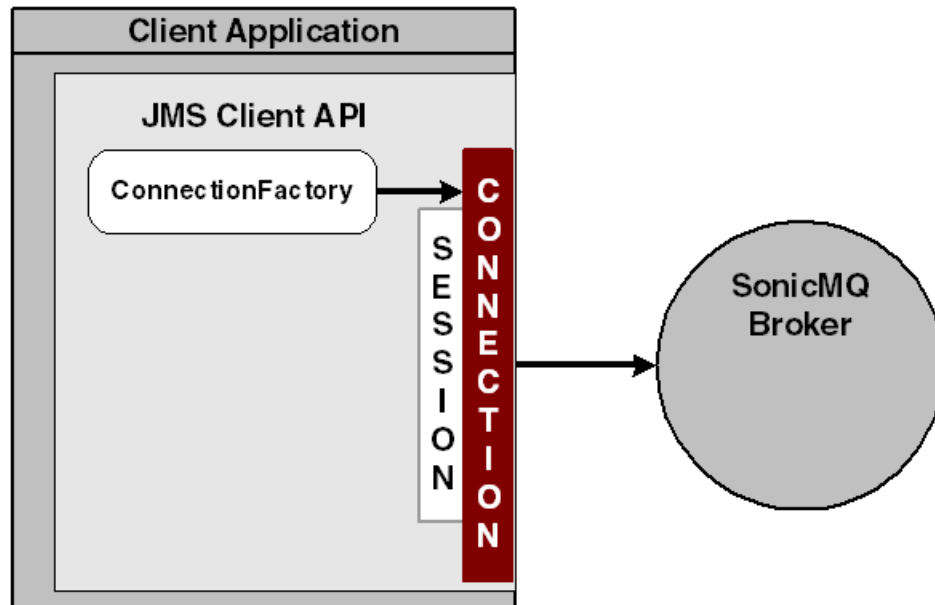
Overview of SonicMQ Connections

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

A SonicMQ application starts by accessing a **ConnectionFactory** and using this to create a **connection** that binds the client to the broker (see [JVM Command Options](#) on page 114). **ConnectionFactory** objects are **administered objects**—objects with connection configuration parameters that can be defined by an administrator (see [Connecting to SonicMQ Using Administered Objects](#) on page 124), or 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. The following figure shows a client application where one connection has been made through which one session has been established. The client application uses programmatic interfaces to the JMS Client API that are executed through the SonicMQ client runtime on the session.

Figure 37: JMS Session on a Connection

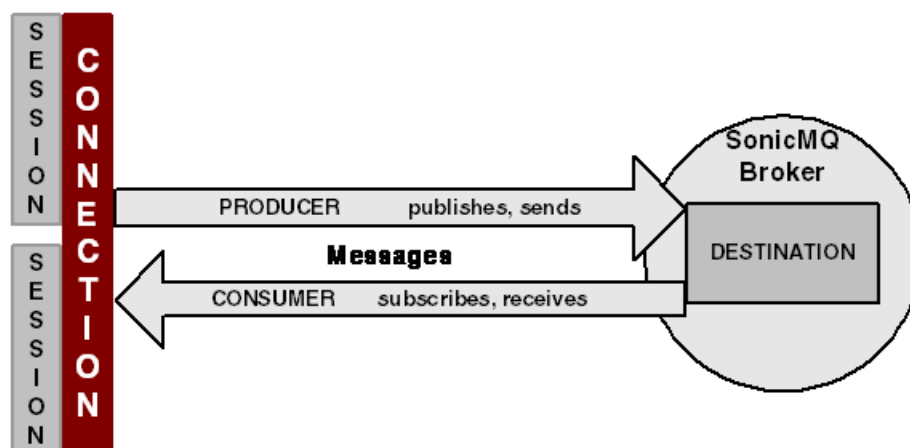


Multiple sessions can be established on a single connection. Once the connection and sessions are established, the broker traffic can be either:

- 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 the following figure, two sessions exist on the same connection.

Figure 38: Producers and Consumers



See [Message Producers and Consumers](#) for more information about message producers and consumers.

Protocols

This section describes the protocols that client applications use for broker communication from a JMS client application:

- [TCP](#) on page 110
- [SSL](#) on page 110
- [HTTP](#) on page 113
- [HTTPS](#) on page 114

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 124 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 Java Secure Socket Extensions (JSSE) SSL. If you have a business arrangement with RSA Security such that you have BSAFE-J SSL libraries from RSA Security, you can add the RSA libraries to your broker installation, and set the broker to use RSA SSL.

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

Using SSL on the Client

Transport layer security between a client application and a broker involves a set of libraries and files on the client and the broker that enable SSL connectivity. Clients need to set several Java runtime properties to identify the SSL provider, the certificates, and the cipher suites preferred for the encryption of the communication channel.

These properties can be established in any one of the following ways:

- **Command line** — Pass the SSL properties at the command line when starting the application.
- **Programmatically** — Code the SSL properties directly into your application.
- **Properties** — Create and reference a properties file containing SSL properties.
- **Scripts** — Create and run a script to pass the SSL properties when starting the application. If you plan to run an application with SSL more than once, you will save time by writing a script to add the properties.

Authentication

Authentication is the process of presenting an identity to the broker and then providing a password or certificate that certifies the user's credentials.

Using Authentication via Username and Password

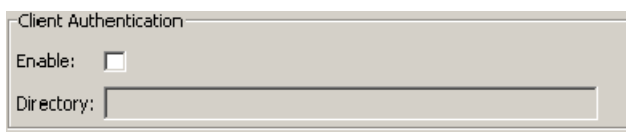
The following procedure explains how to run an application with SSL with client authentication via username and password.

Running the Talk sample application using SSL with password-based client authentication

To run the Talk sample application using SSL with password-based client authentication:

Note: The following steps on the broker use the Management Console. For detailed procedures to perform these steps, see the section “Configuring SSL on Acceptors” in the chapter “Configuring Acceptors” in the *Aurea SonicMQ Configuration and Management Guide*.

1. On the broker:
 - a) Set up or choose an acceptor for SSL connections.
 - b) Clear the option to enable client authentication, as shown in this view from the Management Console:



The screenshot shows a window titled "Client Authentication". Inside, there is a label "Enable:" followed by an unchecked checkbox. Below that is a label "Directory:" followed by an empty text input field.

- c) Set up two users on the broker: **aUser** with the password **aPassword** and **bUser** with the password **bPassword**
2. On the client:
 - a) Open a console window at the directory level of the application you want to run. For example:
 - b) Enter the following code as a single line in the console window:

```
MQ_install_root\samples\QueuePTP\Talk
```

```
..\..\SonicMQ -DSSL_CA_CERTIFICATES_DIR=MQ_install_root\certs\CA
Talk -b ssl://localhost:2506 -u aUser -p aPassword -qr SampleQ1 -qs
SampleQ2
```

The authenticated user is accepted and the application starts.

3. You can send messages between clients using SSL. To demonstrate this process with the **Talk** sample application, you can either administratively add another user to the broker's authentication domain or start another instance of the user already added, as follows:
 - a) Open another command line console window, change directory to the directory level of the **Talk** application, and start the **Talk** application in this window:
 - b) Send messages from each console window and observe the messages as each is received in the other window.

```
..\..\SonicMQ -DSSL_CA_CERTIFICATES_DIR=MQ_install_root\certs\CA
Talk -b ssl://localhost:2506 -u aUser -p aPassword -qr SampleQ2 -qs
SampleQ1
```

Using SSL Authentication via Mutual Certificates

SonicMQ always uses server authentication when using SSL. To have mutual authentication, you must enable Client Authentication on the broker's acceptor. The following procedure explains how to run an application with SSL with client authentication via client certificate.

Running the Talk sample application with client authentication via a client certificate

To run the Talk sample application with client authentication via a client certificate:

Note: The following steps on the broker use the Management Console. For detailed procedures to perform these steps, see the section “Configuring SSL on Acceptors” in the chapter “Configuring Acceptors” in the *Aurea SonicMQ Configuration and Management Guide*.

1. On the broker:
 - a) Set up or choose an acceptor for SSL connections.
 - b) Choose the option to enable client authentication, as shown in this view from the Management Console:



2. On the client:
 - a) Open a console window at the directory level of the application you want to run. For example:

```
MQ_install_root\samples\QueuePTP\Talk
```

- b) Enter the following code as a single line in the console window:

```
..\..\SonicMQ -DSSL_CA_CERTIFICATES_DIR=MQ_install_root\certs\CA
-DSSL_CERTIFICATE_CHAIN=MQ_install_root\certs\client.p7c
-DSSL_PRIVATE_KEY=MQ_install_root\certs\clientKey.pkcs8
-DSSL_PRIVATE_KEY_PASSWORD=password
-DSSL_CERTIFICATE_CHAIN_FORM=PKCS7 Talk -b ssl://localhost:2506 -u
AUTHENTICATED -qr SampleQ1 -qs SampleQ2
```

The connection is authenticated by a mutual exchange of certificates between the client and broker.

Optional:

You can also open another console window at the directory level of the application and start the application by passing the username and password in the command line. This step requires that you have previously added a user with username and password to the broker's authentication domain. For example, if you have added a user with username **bUser** and password **bPassword**, you can enter the following command:

```
..\..\SonicMQ -DSSL_CA_CERTIFICATES_DIR=MQ_install_root\certs\CA
-DSSL_CERTIFICATE_CHAIN=MQ_install_root\certs\client.p7c
-DSSL_PRIVATE_KEY=MQ_install_root\certs\clientKey.pkcs8
-DSSL_PRIVATE_KEY_PASSWORD=password -DSSL_CERTIFICATE_CHAIN_FORM=PKCS7
Talk -b ssl://localhost:2506 -u bUser -p bPassword -qr SampleQ2 -qs
SampleQ1
```

The connection is authenticated by a mutual exchange of certificates between the client and broker, and the broker additionally authenticates the client with the username and password.

Enter messages in both windows, and observe the messages as each is received in the other window.

Now you can open more clients and work with the **Talk** sample application, or implement SSL for other sample applications included with SonicMQ. For each client application, you must either:

- Import a certificate and include the user parameter with username **AUTHENTICATED** when running the sample application.
- Add the user with username and password and provide the password parameter with the password when running the sample application.

Setting Cipher Suites

In SonicMQ, if no cipher suite is specified explicitly, all supported cipher suites are enabled. The client application can provide a subset of the available cipher suites by listing them in the preferred order. For example, using JSSE cipher suites:

```
-DSSL_CIPHER_SUITES=SSL_RSA_WITH_NULL_MD5,SSL_DH_anon_WITH_RC4_128_MD5
```

This statement indicates the following:

- If the broker has the cipher suite **SSL_RSA_WITH_NULL_MD5**, that suite should be used.
- If the broker does not have that suite, the suite **SSL_DH_anon_WITH_RC4_128_MD5** should be tried.
- If neither suite is available, the SSL communication fails regardless of whether the client and server might have compatible cipher suites available in their libraries.

For a list of cipher suite options for SonicMQ, see the section “Cipher Suites” in the chapter “Channel Encryption” in the *Aurea SonicMQ Deployment Guide*.

For information about implementing your own security using the SonicMQ Login SPI, or about using the Login SPI to plug in a Java Authentication and Authorization Service (JAAS) based authentication feature, see the chapter “Security Considerations in System Design” in the *Aurea SonicMQ Deployment Guide*.

HTTP

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. HTTP's other functionality is discussed in the following sections of the *Aurea SonicMQ Deployment Guide*:

- **HTTP Tunneling** — How to set up firewalls and proxy servers is discussed in the *Aurea SonicMQ Deployment Guide* chapter “Security Considerations in System Design.”
- **HTTP Direct** — SonicMQ can interface with pure HTTP Web applications and Web Servers. For example:
 - Inbound to the SonicMQ broker, protocol handlers on acceptors let SonicMQ act as a Web Server, transforming received HTTP documents into JMS messages.
 - Outbound from the SonicMQ broker, sending JMS messages to routing connections that translate the JMS 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 JMS message outbound to the routing node (see page [User-defined Properties](#) on page 211 for details), this book does not discuss the general functionality of HTTP Direct. See the *Aurea 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 *Aurea SonicMQ Deployment Guide* for information about HTTP tunneling.

HTTPS

HTTPS tunneling is similar to HTTP except that data is transmitted over a secure socket layer instead of a normal socket connection. The broker has a different acceptor (configured for HTTPS) than the acceptor that accepts HTTP requests.

Secured HTTP tunneling is discussed in the chapter “SSL and HTTPS Tunneling Protocols” in the *Aurea SonicMQ Deployment Guide*.

HTTPS can be implemented:

- In client-to-broker or broker-to-broker connections
- With or without proxy servers
- Under HTTP forward proxy

JVM Command Options

Several command options can be used in the client application command line, whether it is in a script or an entry line. The following are some of the Java command options available in SonicMQ.

HTTP Tunneling through an Authenticating Proxy

SonicMQ supports three HTTP Authentication schemes for HTTP tunneling connections: Basic, Digest and NTLM. When a proxy presents multiple authentication challenges the client selects the preferred scheme in the following order Digest, NTLM then Basic.

Specifying Credentials

There are several ways to specify the username and password to use for HTTP authentication. (When running from an applet, this is not necessary as the browser handles HTTP credentials.)

- By setting the following System properties:
 - Dsonic.http.proxyUsername=username
 - Dsonic.http.proxyPassword=password
- By setting the following System property:
 - Dsonic.http.authenticator=className

where **className** is the name of an accessible class that provides a concrete implementation of **java.net.Authenticator**.

- By registering an instance of a **java.net.Authenticator** via **java.net.Authenticator.setDefault(java.net.Authenticator)**

NTLM Authentication

A client can perform NTLMv1 authentication if a proxy requests it. For regular Java applications SonicMQ NTLM authentication is supported on all platforms. When running from an Applet NTLM authentication is currently only supported on Windows machines and it is up to the browser plugin to handle NTLM authentication.

When NTLM is used as the authentication scheme a Windows domain name must be provided. This can be done in one of two ways:

- By prepending the domain to the username separated by a backslash, as shown:

```
DOMAIN\<username>.
```

This method takes precedence over the following one.

- By setting the following System property:

```
-Dhttp.auth.ntlm.domain=domain
```

where **domain** is the Windows domain name.

NTLM authentication also requires that a workstation name be provided. By default the machine's hostname is used if it can be obtained, otherwise it can be specified explicitly by setting the following System property:

```
-Dsonic.http.auth.ntlmWorkStation=workstation
```

HTTP Forward Proxy

In order to configure an HTTP tunneling client through a forward proxy, the following standard JDK properties are supported to specify the forwarding proxy host:

```
-Dhttp.proxyHost=proxy_host_name  
-Dhttp.proxyPort=proxy_host_port
```

HTTPS Forward Proxy

In order to configure an HTTPS tunneling client through a forward proxy, the following SonicMQ properties are supported to specify the forwarding proxy host:

```
-Dhttps.proxyHost=proxy_host_name  
-Dhttps.proxyPort=proxy_host_port
```

HTTPS Tunneling Through an Authenticating Forward Proxy

If you want to tunnel through a secure forward proxy server using HTTPS, use the following procedures for authentication:

Note: Proxy authentication requires that the client uses Sun JVM 1.4.1_02 or similar.

1. Enable tunneling via a secure proxy

Set the system property **-Dsonic.https.proxyAuthentication** on the client's Java command line to enable tunneling through a secure proxy using **javax.net.ssl.HttpsURLConnection**. Since the JVM's **HttpsURLConnection** class uses JSSE, the required CA certificates must be imported to a **trustStore**.

If client authentication is enabled on the broker, the client certificate and its corresponding private key must be imported to a **keyStore** as well.

See Sun's JSSE documentation for more details.

2. Register an Authenticator

A concrete subclass of **java.net.Authenticator** is required to handle proxy authentication. Applications register an authenticator programatically using the static method **setDefault** of the **java.net.Authenticator** class.

Instead, you can direct the Sonic runtime to install an authenticator by specifying the package qualified class name as the **-D** system property **sonic.https.proxyAuthenticator** on the client's Java command line.

A default authenticator for BASIC authentication is provided if the system properties **sonic.https.proxyUsername** and **sonic.https.proxyPassword** are specified.

3. Register a Hostname Verifier

A concrete subclass of **javax.net.ssl.HostnameVerifier** is required to register a hostname verifier. An application can register a hostname verifier programatically using the **setHostnameVerifier** method of the **HttpsURLConnection** class.

Instead, you can direct the Sonic runtime to install a hostname verifier by specifying the **-D** system property **sonic.https.hostnameVerifier** on the client's Java command line.

A default hostname verifier that accepts any hostname in the certificate is provided if the **-D** system property **sonic.https.useAnyHostnameVerifier** is specified.

SSL/HTTPS

The following SSL command options were shown in the procedure for running the **Talk** sample application with client authentication via a client certificate:

```
-DSSL_CA_CERTIFICATES_DIR=MQ_install_root\certs\CA
-DSSL_CERTIFICATE_CHAIN=MQ_install_root\certs\client.p7c
-DSSL_PRIVATE_KEY=MQ_install_root\certs\clientKey.pkcs8
-DSSL_PRIVATE_KEY_PASSWORD=password
-DSSL_CERTIFICATE_CHAIN_FORM=PKCS7
```

Nagle Algorithm

The Nagle algorithm allows buffering of small data before sending the data as a fully constructed IP packet. By default, this algorithm is disabled.

To enable this algorithm, set **-DSonicMQ.TCP_NODELAY=false** on the JVM command line; to disable it, set **-DSonicMQ.TCP_NODELAY=true**.

HTTP Map Host to IP

This client setting indicates whether conversion of the host name to its corresponding IP Address should be attempted before connecting. In some environments, the client system does not have a DNS available but the forward proxy server system does. When this property is set to **false**, the HTTP requests are sent from the client to the forward proxy, with the **HOST** header set to the host name instead of the host's IP address. This allows the DNS lookup to be delayed until the proxy server tries to establish the connection to that host.

The syntax of the property is:

```
-DHTTP_MAP_HOST_TO_IP=[true|false]
```

where:

- **true** causes conversion of the host name to its IP address before connecting (this is the default value)
- **false** causes no conversion of the host name to its IP address before connecting

Connection Factories and Connections

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

Connection Factories

To establish a Java connection with the SonicMQ broker, a Java client uses a **ConnectionFactory** object. Prior to JMS 1.1, model-specific factories were required for the Pub/Sub and Point-to-Point message models; however, beginning in JMS 1.1, common connection factories can be used for both models.

These common connection factories are:

- **ConnectionFactory**
- **XAConnectionFactory**

Java clients can obtain a connection factory in the following three ways:

- Instantiating a new connection factory object by specifying connection information in the object constructor (and possibly customizing further using set methods on the factory)
- Obtaining a preconfigured connection factory object from a JNDI store
- Deserializing a preconfigured factory object from a file

Each of these techniques is described in this chapter.

SonicMQ connection factory objects encapsulate the information needed to connect and configure the SonicMQ JMS 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 JMS client registrations. These identifiers have specific name restrictions, shown in the following table.

Important: The following table lists characters that are not allowed in SonicMQ. You must not use these restricted characters in your identifier names. See also Appendix A of *Aurea Sonic Installation and Upgrade Guide* for a complete reference to use of characters in SonicMQ names.

Table 4: Restricted Characters for Names

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

Note: Although a Durable Subscription name is not a connection factory setting, it is included in the above table for completeness.

URL

The Uniform Resource Locator 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 on the host where the broker is listening. The broker's 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 some additional identifier.

Note: See [Connection Factories](#) on page 117 for a list of restricted characters for ConnectID names.

ConnectID can also be preconfigured in a **ConnectionFactory** administered object, or passed as an argument to a SonicMQ **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:

- Preconfigured in a **ConnectionFactory** administered object
- 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 [Connection Factories](#) on page 117 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, do one of the following:

1. In the client application, immediately after creating a connection, call the **Connection.setClientID(String clientid)** method.
2. Set the **ClientID** in the **ConnectionFactory**. You can either preconfigure the **ClientID** via the JMS Administered Objects tool in the Sonic Management Console, or you can call **ConnectionFactory.setClientID(String clientid)** in the client application.

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

Note: See [Connection Factories](#) on page 117 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.

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.

Note: When using custom load balancers on the broker, you can provide hints to the broker by using the method **setLoadBalancingClientData(String clientData)** in the Java client and then using **getClientData()** in the load balancer.

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

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 will attempt to connect to brokers in this list.
2. Call **ConnectionFactory.setConnectionURLs(brokerList)** to point to the text list you created. The client will connect 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, call **ConnectionFactory.getConnectionURLs()** to return the broker list, and then 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, as shown:

```
javax.jms.ConnectionFactory factory;  
factory = (new progress.message.jcclient.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.

This option can be set on Connection Factories that are defined as Administered Objects. See the *Aurea SonicMQ Configuration and Management Guide* for information.

Setting a Socket Connect Timeout

You can specify a timeout to be used when establishing a socket connection to a broker. The **ConnectionFactory** method **setSocketConnectTimeout (int timeout)** lets you set the number of milliseconds to allow for the socket connection to be established, as shown:

```
javax.jms.ConnectionFactory factory;  
factory = (new progress.message.jcclient.ConnectionFactory (broker));  
factory.setSocketConnectTimeout(5000);  
connect = factory.createConnection (username, password);
```

Setting a value of 0, the default value, means the socket connect request does not time out.

If the socket connection is not established within the timeout interval, an exception is returned to the caller with the error code **ERR_SOCKET_CONNECT_TIMEOUT**.

Note: The **SocketConnectTimeout** setting interacts with the **InitialConnectTimeout** setting described in [Specifying Connection Timeouts](#) on page 143, and—for fault tolerant connections—the operating systems settings discussed in the “Tuning TCP to Optimize CAA Failover” in the *SonicMQ Performance Tuning Guide*. The socket connect timeout should enable an attempt at every listed URL. For example, where a URL list contains six URLs, the default setting for the **InitialConnectTimeout** of 30 seconds would require that the **SocketConnectTimeout** value be set to 5 seconds. The tuning of the operating system for fault tolerant failover assures that the OS does not add unintended delays.

This option can be set on Connection Factories that are defined as Administered Objects. See the *Aurea SonicMQ Configuration and Management Guide* for information.

Setting QoP Cache Size

Brokers that enable security and Quality of Protection (QoP) require that clients enforce (or override) the QoP setting specified for each destination when sending messages. Whenever a broker responds to a client with the appropriate setting, the client caches the values in its QoP cache, adequate for 128 QoP settings for topics and queues as well as other connection administration, actions, sessions, and message producers/consumers. Using a least-recently-used algorithm for clearing the cache so that it can accommodate new entries, the cache is adequate and efficient in most situations. However some circumstances make it crucial to increase the size of the cache so that the cache is not constantly being updated.

When using MultiTopics (see [MultiTopics](#)), the topic list might easily surpass the client connection's cache limit. If this situation occurs, every topic is sent with QoP set to **PRIVACY**, and the response from the broker indicates whether that level of protection was required. That QoP setting is cached but might be promptly dropped when other QoP settings are recorded in the cache. In that case, you can modify the cache size through the **ConnectionFactory** parameter:

```
ConnectionFactory.setQopCacheSize(Integer size)
```

where **size** needs to accommodate the application topics yet leave space for other cached items. The recommended value when you choose to reset the QoP cache is:

(the number of application topics or queues) + 128

For example:

```
javax.jms.ConnectionFactory factory;
factory = (new progress.message.jclient.ConnectionFactory (broker));
factory.setQoPCacheSize(256);
connect = factory.createConnection (username, password);
```

This option cannot be set on Connection Factories that are defined as Administered Objects.

Setting the Maximum DeliveryCount

The **setMaxDeliveryCount** method in **progress.message.jclient.ConnectionFactory** sets the maximum number of times delivery of a message to a consumer should be attempted. Messages that have exceeded the delivery limit are processed according to message properties that govern disposition of undeliverable messages.

The following syntax sets the maximum delivery count:

```
ConnectionFactory.setMaxDeliveryCount(java.lang.Integer value)
```

where:

- **value** is **0** when you want no redelivery limit
- **value** is an positive integer that specifies to deliver and then redeliver the specified number of times

A related method is **public java.lang.Integer getMaxDeliveryCount()**. It returns the integer value set (or defaulted) for the maximum delivery limit.

For example:

```
javax.jms.ConnectionFactory factory;
factory = (new progress.message.jclient.ConnectionFactory (broker));
factory.setMaxDeliveryCount(10);
connect = factory.createConnection (username, password);
```

This option can be set on Connection Factories that are defined as Administered Objects. See the *Aurea SonicMQ Configuration and Management Guide* for information.

Setting to Minimize Subscriber Traffic

The **setMinimizeSubscriberTraffic** method in **progress.message.jclient.ConnectionFactory** provides control over **TopicSubscribers** and **DurableSubscribers**. When set to **true**, the subscriber will attempt to flow control the broker as soon as messages are delivered into the client's buffer. The subscriber could receive more messages put on the wire by the broker before it receives the flow control message. Sending resumes when the subscriber's buffer becomes empty.

The feature lets an applications effectively shut down or reduce the subscriber's client buffer to minimize the background priming (at the cost of increased latency).

The following syntax sets the option to minimize subscriber traffic:

```
ConnectionFactory.setMinimizeSubscriberTraffic(boolean value)
```

where:

- **value** is **true** when you want to minimize subscriber traffic

A related method is **public java.lang.boolean getMinimizeSubscriberTraffic()**. It returns the value that indicates whether subscriber traffic is being minimized.

This option can be set on Connection Factories that are defined as Administered Objects. See the *Aurea SonicMQ Configuration and Management Guide* for more information.

Note: Feature in C and C++ clients — This feature is also in the Sonic C and C++ clients. In those clients, the option is set in a parameter of a new **ConnectionFactory** signature. See the *Aurea SonicMQ C Client Guide* and *Aurea SonicMQ C++ Client Guide* for more information.

Enabling Message Compression

Some throughput problems are caused by large messages and low bandwidth networks. Applications that produce and consume messages of significant size over these slow networks might improve overall performance by compressing messages.

The **setEnableCompression** method in **progress.message.jclient.ConnectionFactory** causes messages produced in the scope of the connection factory to be compressed so that:

- **MessageProducers** compress every message before sending it, and the broker decompresses every message it receives on these connections.
- **MessageConsumers** decompress every message when received because the broker compressed every message it delivered to the consumer on these connections.

When a SonicMQ client application enables message compression, the client negotiates with the broker to which it is connecting to agree on the compression characteristics and error checking. The actual compression and decompression functions are implicit when the option is enabled.

Message compression has time and space requirements on both the client and the broker. An administrator needs to determine which connections can offset the compression overheads with the savings in message transfer time, and how many connections that enable compression can be supported by the broker's resources.

The following syntax enables message compression on a connection factory:

```
ConnectionFactory.setEnableCompression(boolean value)
```

For example:

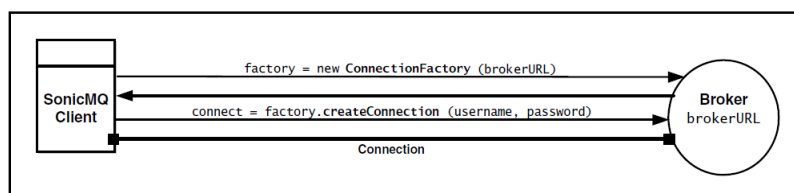
```
javax.jms.ConnectionFactory factory;
factory = (new progress.message.jclient.ConnectionFactory (broker));
factory.setEnableCompression(true);
connect = factory.createConnection (username, password);
```

This option can be set on Connection Factories that are defined as Administered Objects. See the *Aurea SonicMQ Configuration and Management Guide* for information.

Connecting to SonicMQ Directly

An application can use the SonicMQ API directly to create a new **ConnectionFactory** object, as shown in the following figure. 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 39: 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 **javax.jms.JMSSecurityException** is thrown. You can use the common name from a certificate when you use SSL mutual authentication. See the *Aurea SonicMQ Deployment Guide* for more about SSL and security.

Connecting to SonicMQ Using Administered Objects

JMS administered objects are objects containing JMS configuration information that are created by a JMS administrator and later used by JMS clients. These objects make it practical to administer JMS applications in the enterprise.

JMS specifies the following types of administered objects:

- Connection Factories:
 - **ConnectionFactory** and **XAConnectionFactory**
 - **QueueConnectionFactory** and **XAQueueConnectionFactory**
 - **TopicConnectionFactory** and **XATopicConnectionFactory**

Note: The JMS 1.1 specification states that some JMS version 1.02b (model-specific) interfaces might be deprecated in the future. Consequently, if you are developing new JMS client applications,

it is recommended that, wherever possible, you use the common interfaces in place of the older model-specific interfaces. Here, you should use **ConnectionFactory** and **XAConnectionFactory** instead of the model-specific interfaces.

-
- Destinations
 - Queue
 - Topic

JMS client applications obtain instances of SonicMQ connection factory objects (see [Lookup Using the Sonic JNDI SPI](#) on page 126) and use JMS specified factory methods on those objects to create connections. (See [Lookup and Use of Administered Objects](#) on page 125.)

Important: Permission Denied Issues for Older Clients — If you are using JNDI SPI clients and your domain enforces management permissions (a feature introduced in V7.5), the JNDI SPI clients should be upgraded to at least V7.5 to avoid the potential of spurious **ConfigurePermissionsDenied** exceptions which could deny JNDI access.

Advantages of Using JMS Administered Objects

JMS administered objects can be created using tools provided in SonicMQ (see the *Aurea SonicMQ Configuration and Management Guide* for information on using the Management Console to create JMS administered objects). Administered objects hide vendor-specific information. Since administered objects implement a public interface and can be retrieved using JNDI, JMS client applications can be coded to be independent of JMS vendor implementations.

The indirection the JNDI lookup name provides has an additional and more significant benefit: JMS client applications can be coded to be independent of broker location. For example, the application can be coded to use a factory located under the name **cn=QCF**, without knowing which broker will service the application. When some deployment change is made (for example, when a backup system comes online or if during certain hours load is directed to another machine), the administrator simply replaces the connection factory stored at the location **cn=QCF** with another factory instance that encapsulates connection information to a broker running on a different system.

Lookup and Use of Administered Objects

SonicMQ administered objects are both **serializable** and **referenceable**, and thus can be stored in a wide range of JNDI accessible stores, including Sonic's own internal JNDI store and LDAP. SonicMQ provides tools and APIs with which to create, store and lookup SonicMQ implementations of administered objects.

This chapter describes how to use the following JNDI SPIs:

- In the SonicMQ internal JNDI store
- In an external LDAP server through JNDI

See the *Aurea SonicMQ Configuration and Management Guide* for information about using the Sonic Management Console. See [Appendix 4, Using the Sonic JNDI SPI](#) for information about programming using the JNDI SPI.

In the code samples that follow, the name used to find an administered object is formatted to correspond to the store implementation used to store the object:

- Simple name: **ContextName** — For the SonicMQ internal JNDI store.
- Schema name: **cn=ContextName** — For an external LDAP server through JNDI.
- Filename: **ContextName.sjo** — For a serialized file object.

Lookup Using the Sonic JNDI SPI

JNDI defines the way an initial context is obtained; obtaining a Sonic context follows these same techniques. The **JNDITalk** sample (an excerpt of which is shown in the following sample code) provides a simple demonstration of JNDI programming with the Sonic SPI. The sample shows:

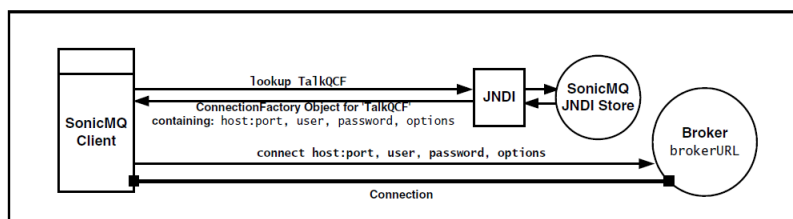
- Creating a JNDI environment (hash table) with Sonic SPI specific values and additional properties
- Obtaining an initial context
- Using the context to perform a lookup of a **ConnectionFactory** object

Programming with the Sonic JNDI SPI (JNDITalk Sample)

```
private static final String QCF_LOOKUP_NAME = "TalkQCF";
...
Context context = null;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sonicsw.jndi.mfcontext.MFContextFactory");
env.put(Context.PROVIDER_URL, "tcp://localhost:2506");
env.put("com.sonicsw.jndi.mfcontext.domain", "Sonic");
env.put("com.sonicsw.jndi.mfcontext.idleTimeout", "60000");
env.put(Context.SECURITY_PRINCIPAL, "Administrator");
env.put(Context.SECURITY_CREDENTIALS, "Administrator");
...
context = new InitialContext(env);
...
javax.jms.ConnectionFactory factory = null;
...
factory = (javax.jms.ConnectionFactory) context.lookup(QCF_LOOKUP_NAME);
...
```

This type of lookup submits a name to the JNDI store for lookup. In the following figure, the factory name **TalkQCF** (a simple arbitrary name for a **ConnectionFactory** object used in these examples) is submitted in the format **TalkQCF**.

Figure 40: Connecting to SonicMQ Using JNDI



Note: The context name can also be submitted in the LDAP format: **cn=TalkQCF**, but this format is not required.

Setting up an Administered ConnectionFactory Object

The **JNDITalk** example attempts to lookup a **ConnectionFactory** object. For the lookup to succeed, the administered **ConnectionFactory** object must be defined in the Sonic JNDI store. You can define and store an administered **ConnectionFactory** object using the JMS Administered Objects tool. The *Aurea SonicMQ Configuration and Management Guide* provides detailed instructions on the JMS Administered Objects tool.

To create an administered ConnectionFactory object for the JNDITalk sample:

1. Start the SonicMQ Container that hosts the Directory Service that the broker will use.
2. Start the Sonic Management Console.
3. Select **Tools > JMS Administered Objects**. Select the local JNDI store:
 - a) Choose JNDI Naming Service.
 - b) Select the Sonic Storage option.
 - c) Enter the URL for the Directory Service container.(**localhost:2506** for example.)
 - d) Enter the user, such as **Administrator**.
 - e) Enter the user's password, such as **Administrator**.
 - f) Click **Connect**.

The provider URL you entered appears in the **Object Stores** list in the **JMS Administered Objects** window, and a node for this provider URL appears in the left panel.

4. Set up the connection factory. For the example:
 - a) In the left panel of the Sonic Management Console, choose the connection you just established to the JNDI Naming Service.
 - b) In the right panel, choose the **Connection Factories** tab then click **New**.
 - c) In the **Lookup Name** field, enter a new record with **TalkQCF** as the name value.
 - d) From the **Factory Type** pull-down list, choose **ConnectionFactory**.
 - e) Enter an **URL** for the application connection, such as **localhost:2506**

Do not enter a user or password. The example will override the username and password and show how they can be supplied in application parameters, thus enabling varied authorizations for applications that use the lookup information.

- f) Enter a **Connect ID** such as **First**. This is a value that will be changed in the example to demonstrate how administrative changes to the lookup value are passed through the connections that use the connection factory.
- g) Click **Update**.

The **TalkQCF** object is entered in the JNDI store.

Running the JNDITalk Sample

You are now ready to run the modified **Talk** sample that performs a lookup to the JNDI store to get a context.

To run the JNDITalk sample, do the following:

1. In a console window at the **JNDITalk** directory, enter: `..\..\SonicMQ JNDITalk -u Administrator -p Administrator -qr SampleQ1 -qs SampleQ2`
2. Open a command line console window, change directory to the **Talk** directory then enter: `..\..\SonicMQ Talk -u Administrator -p Administrator -qr SampleQ2 -qs SampleQ1`

Each sample will receive the messages sent by the other application.

3. In the **Talk** console window, type some text and press **Enter**.

The **JNDITalk** window displays the text, preceded by:

Administrator:

4. In the **JNDITalk** window, enter text and press **Enter**.

The **Talk** window displays the text preceded by:

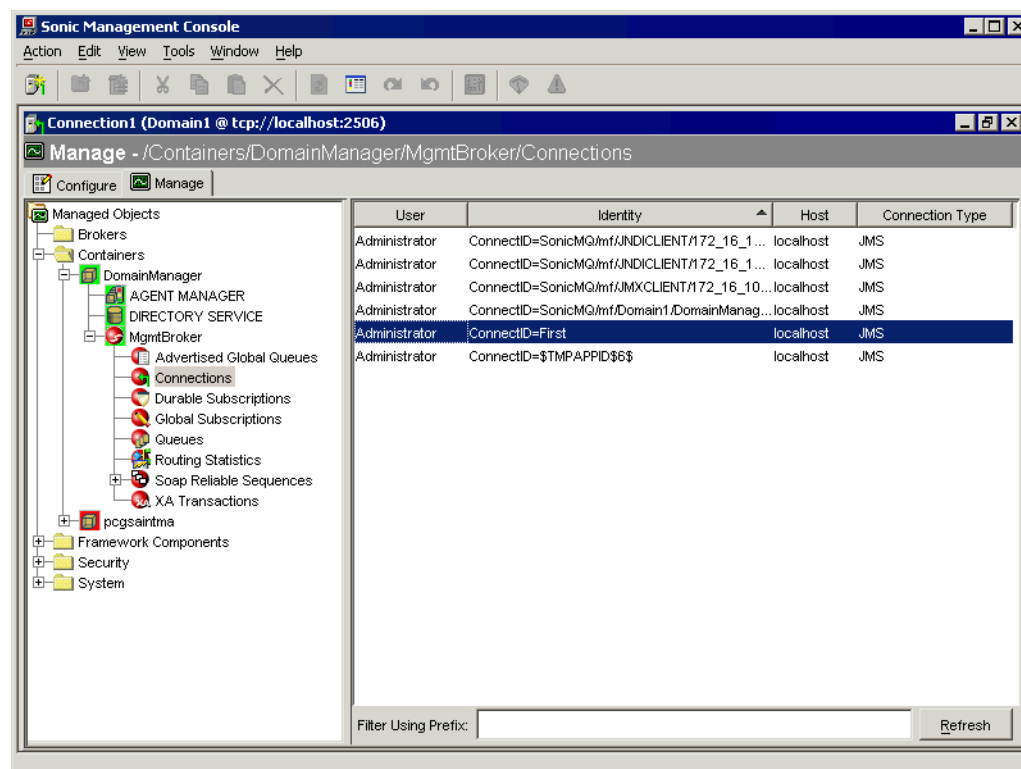
Administrator:

You can extend this test by looking at the connection through the Sonic Management Console:

- Under the **Manage** tab, in the left panel click **Containers\Container1**, then click the node for your broker. Click the **Connections** node under the broker.

The right panel lists the connections for this broker, as shown in the following figure:

Figure 41: Connection Using JNDI Store Lookup



One of the connections lists its Connect ID as **First**, the name used for the **ConnectionFactory** stored in the JNDI store.

If you use the JMS Administered Objects window to update the **TalkQCF** object to have a Connect ID of **Next**, that value will not be reflected in the connections until the connection factory is looked up again. By stopping the **JNDITalk** application and then restarting it, the connection listed in the Management Console will display the revised Connect ID for **TalkQCF**.

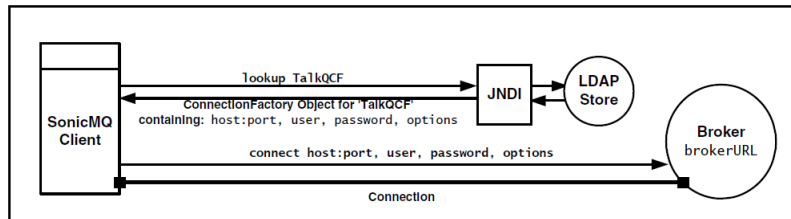
Using the LDAP JNDI SPI

JNDI provides interfaces to standard directory servers such as those that are compliant with the Lightweight Directory Access Protocol (LDAP). With SonicMQ, you can either use the internal JNDI package provided with SonicMQ to access objects stored in the SonicMQ directory server, or you can access an external LDAP directory server, as described in this section.

Important: External LDAP directory servers are distinct products that you must install and configure separate from SonicMQ. The Javasoft JNDI Web site can point you to evaluation editions of LDAP directory servers so that you can explore these services.

In the following figure, the context name **TalkQCF** is submitted as **cn=TalkQCF**.

Figure 42: Alternate Connection Techniques Using Factory Objects or JNDI Lookup



From a client program, select an external LDAP server such as the JNDI store by setting the system property “**javax.naming.Context.INITIAL_CONTEXT_FACTORY**” to “**com.sun.jndi.ldap.LdapContextFactory**”. The property “**javax.naming.Context.PROVIDER_URL**” specifies how to locate to LDAP server and establish the initial JNDI naming context. For example: “**ldap://myipc.a.sonicmq.com:389/ou=jmsao,ou=sonicMQ,o=a.sonicmq.com**”

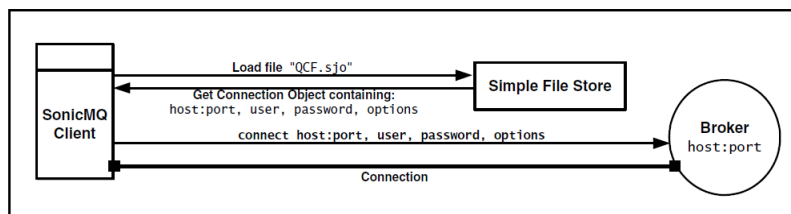
See [Java JNDI SPI Sample](#) for information about a sample application that demonstrates using the Sonic JNDI SPI.

Connecting to SonicMQ Using Serialized Factories

SonicMQ allows you to administratively store objects as Serialized Java Objects (**.sjo**) in a file system. By updating the **.sjo** objects with the JMS Administered Objects tool in the Sonic Management Console, you can isolate the programmer from specific broker configuration parameters and destination names. However, the programmer must still maintain and deploy the **.sjo** files.

The following figure illustrates deserializing a factory from a specified path location. An administrator stores serialized Java objects as files with **.sjo** extensions. The files can then be loaded (and deserialized) using the **java.io** package.

Figure 43: ConnectionFactory Object Instantiated By Lookup of a Serialized Java Object



Setting Up Serialized Objects

The following code provides an example of how serialized objects can be set up. This example assumes:

- The **ConnectionFactory** for the sample application is stored in the file **ChatConnectionFactory.sjo**.
- The **Topic** for the application is stored in the file **ChatTopic.sjo**.
- A new method, **readFile()**, is used for both administered objects.

readFile Method

```
/**
 *Read an object from the given file.
 *@param filename The name of the file.
 *@return The deserialized object. If the file does not contain a valid JMS managed
 *object
 *or there is some read/deserialization problem, then return null.
 */
private Object readFile(String filename)
{
    try
    {
        java.io.FileInputStream fis = new java.io.FileInputStream(filename);
        java.io.ObjectInputStream ois = new java.io.ObjectInputStream(fis);
        Object readObj = ois.readObject();
        fis.close();
        return readObj;
    }
    catch(java.io.IOException e) { } // return null
    return null;
}
```

Using Serialized Objects

After serialized objects are set up, those objects can be used in your applications. Within the application code where the connection is established, use the **readFile()** method to read the active **javax.jms** objects, as shown in the following code.

Using readFile to Read Active javax.jms Objects

```
javax.jms.ConnectionFactory factory;
// Read in the factory from a file
factory = (javax.jms.ConnectionFactory) readFile("ChatConnectionFactory.sjo");
...
// Continue, creating connection from the factory
// Continue, creating the session from the connection.
...
// Finally, retrieve the TOPIC for our application
javax.jms.Topic topic = (javax.jms.Topic) readFile ("ChatTopic.sjo");
```

Connections

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

Important: The JMS specification states that an application should not use a Java constructor to create connections directly, otherwise applications will not be portable.

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, the threads used for producing and consuming messages.

You create **Connection** 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, or when your JNDI store is very secure.

- Supply the preferred username and its authenticating password:

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

Creating and Monitoring a Connection

The following code snippet, 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.

ReliableChat: setupConnection

```
// Get a connection factory
javax.jms.ConnectionFactory factory = null;
try
{
    factory = (new progress.message.jclient.ConnectionFactory(m_broker));
} catch (javax.jms.JMSEException jmse) ...
//Wait for a connection.
while (connect == null)
{
    try
    {
        System.out.println("Attempting to create connection...");
        connect = (progress.message.jclient.Connection)
            factory.createConnection (m_username, m_password);
        ...
        //Ping the broker to see if the connection is still active.
        connect.setPingInterval(30);
    }
    catch (javax.jms.JMSEException jmse)
    {
        System.out.print("Cannot connect to broker: " + m_broker );
        System.out.println("Pausing" + CONNECTION_RETRY_PERIOD / 1000 + " seconds before
        retry.");
        try
        {
            Thread.sleep(CONNECTION_RETRY_PERIOD);
        }
        catch (java.lang.InterruptedExcepion ie) { }
        continue;
    }
    ...
}
```

The statement **connect.setPingInterval(30)** indicates the use of a method that lets the application detect when a connection gets dropped by setting a **PingInterval** of 30 seconds. The **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 is required to respond to each ping sent by the client. If the client does not receive any traffic within a ping interval, then the client assumes the connection is bad and drops the connection. See [Handling Exceptions on the Connection](#) on page 132 for more information. But 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 will not cause a connection drop. See [Fault-Tolerant Connections](#) on page 140 for additional information about fault tolerant connections.

You can also configure active pings in the **ConnectionFactory** by invoking the **ConnectionFactory.setPingInterval(interval_in_seconds)** method, or by preconfiguring **ConnectionFactory** administered objects with a ping interval.

Synchronous pings are always used when you explicitly set a non-zero ping interval. The difference between fault tolerant and non-fault tolerant is when the ping interval is not set. There are no pings by default on a non-fault tolerant connection, whereas there are asynchronous pings by default on a fault tolerant connection.

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

Handling Exceptions on the Connection

The exception handler can handle errors actively as shown in the following code snippet, from the **ReliableChat** sample, where a connection problem initiates a reconnection routine.

ReliableChat: Reconnection Routine

```
// Handle asynchronous problem with the connection.
public void onException ( javax.jms.JMSEException jsme)
{
    // See if connection was dropped.
    //Tell the user that there is a problem.
    System.err.println("\n\nThere is a problem with the connection.");
    System.err.println("    JMSEException: " + jsme.getMessage());
    //If the error is a dropped connection, try to reconnect.
    // NOTE: the test is against Aurea SonicMQ error codes.
    int dropCode = progress.message.jclient.ErrorCodes.ERR_CONNECTION_DROPPED;
    if (progress.message.jclient.ErrorCodes.testException(jsme, dropCode))
    {
        System.err.println ("Please wait while the application tries to "+ "re-establish
the connection...");
        // Reestablish the connection
        connect = null;
        setupConnection();
    }
}
```

Handling Dropped Connection Errors

When broker failure occurs, the existing protocol reset initiates the **onException()** method of the **ExceptionListener** with the error code:
progress.message.jclient.ErrorCodes.ERR_CONNECTION_DROPPED.

In the case of network failure, when a broker becomes disconnected from the network JMS clients generally notice some time after they try to publish or send a message. If the application is only acting as a subscriber, network failure might not be detected by the client. Enabling active ping will ensure timely detection of loss of network.

Exception Listeners Are Not Intended for JMS 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 exceptions delivered to **ExceptionListener** are those that do not have any other place to be reported. If an exception is thrown on a JMS call, then by definition the exception must not be delivered to an **ExceptionListener**. In other words, the **ExceptionListener** is not for the purpose of monitoring all exceptions thrown by a connection.

Client Persistence

SonicMQ installations that provide ClientPlus features have the option of enabling client persistence. Client persistence provides a higher level of reliability than is defined in the JMS specification. Where a network failure during a JMS send would normally cause a message being sent to be effectively lost unless the user application takes additional precautions, client persistence enables client-based logging of messages sent until the broker connection is re-established. This feature enhances delivery guarantees and provides disconnected operation.

When flow control forces a message producer to pause, clients that have enabled client persistence continue to produce messages into the persistent store. When producer flow control is no longer in effect, persisted messages flow to the broker in order while the message producer continues to add messages to the local store. When the local store is cleared, messages flow directly from the producer to the broker.

The persistent store is a set of files in a directory name specified by the user in association with a JMS connection. The client run time uses the files to store messages and manage their delivery to the SonicMQ broker.

The characteristics of the client persistence store and the wait time before flow controlled messages are persisted in the store can be set programmatically on the connection factory or on connection factory administered objects.

For each connection using persistence, the persistence directory on the client includes a subdirectory identified by **ClientId** that contains:

- One or more files to store messages rejected by the broker
- Recovery files for logging restart information
- For each session on the connection, a file that records all messages sent while in the disconnected state

When the client has an active connection to the broker, the client operates normally. Messages are not written to the persistent store until a network outage or a flow control pause is detected.

When a network outage or a flow control pause is detected, the currently active message is written to the persistent store and the client switches to writing all messages—persistent and non-persistent—to the store. The size of the session log is limited by the local store size. Non-persistent messages in memory when the outage is detected are dropped.

While disconnected because of a network outage, the client runtime tries to reconnect to a broker. It is possible for the client to reconnect to a different broker (than it originally connected to) if you provided a list of brokers in the connection parameters of the factory, or if broker load balancing is configured and the client elected load balancing.

When a connection is reestablished or the flow control is no longer in effect, the client runtime sends all persistent messages in memory at the time of the disconnect, then replays the session log. New messages are accepted while the messages in the session log are sent to the broker and acknowledged. The persistent client controls the rate of accepting messages into the store relative to the rate of sending stored messages out of the local store to the broker in an effort to drain the backlog of messages. The sender experiences a slower producer rate while messages are being restored. However, it is possible for messages to accumulate in the store faster than they can be sent to the broker. If this occurs, the local store size might be exceeded in which case the sender gets an exception.

Files are deleted after all messages have been sent to the broker and acknowledged and all rejections have been processed by the **RejectionListener**. An application should explicitly close sessions and connections to allow the client runtime to perform cleanup. In the event of an abnormal end to the client connection, the next startup will send unacknowledged messages and cleanup unneeded files.

Using Client Persistence

SonicMQ applications that want to use client persistence require some modest coding changes from existing SonicMQ applications. Because this feature is asynchronous store-and-forward, immediate feedback on delivery failures is not available. To compensate, the application must set up a listener to handle send failures. Also JMS functions such as creation of receivers and transacted sessions are not allowed when using client persistence (see [Coding Limitations](#) on page 135). The directory and size of the persistent store is specified on the **ConnectionFactory**.

The following code snippet, from the **ContinuousSender** sample application, shows a coding construct for implementing client persistence.

Continuous Sender: Implementing Client Persistence

```
//Connect id is required when using local Store
    factory = (new progress.message.jclient.QueueConnectionFactory
(m_broker,"StoreTest"));
//Configure factory for local store
// ClientId must be set in the factory when using the local store
    factory.setClientID(CLIENT_ID);
    factory.setEnableLocalStore(true);
    factory.setLocalStoreDirectory("MyDir");
    factory.setLocalStoreSize(1000);
// 1 MB
// seconds before client persists when flow controlled
    Integer waitPersist = new(Integer(5));
    factory.setLocalStoreWaitTime(waitPersist);
    connect = (progress.message.jclient.QueueConnection)
factory.createQueueConnection(m_username, m_password);
//Enable client ping to expedite loss of network detection
// on some UNIX platforms
    connect.setPingInterval(30);
    connect.setRejectionListener (this);
```

This sample accepts the default values of the parameters for reconnect timeout and reconnect interval. The parameters in this example are similarly used for **TopicConnectionFactory** in the **ContinuousPublisher** sample.

The setter methods for client persistence are listed in the following table:

Table 5: ConnectionFactory Methods for Client Persistence (Continued)

Method	Default	Description
setEnabledLocalStore(boolean value)	false	Enables use of the local store.
setLocalStoreDirectory(String name)	current working directory	The name of the directory that the local stores for the connection factory will use. When you have multiple connection factories, use different directories to avoid unpredictable behaviors.
setLocalStoreSize(long size)	10000 (10MB)	The maximum size of the local store (in Kilobytes). The size puts a limit on how many messages can be stored while operating in disconnected mode.
setReconnectTimeout(int minutes)	0 (none)	Sets how long (in minutes) the runtime should try to make a connection to the broker at which point an exception will be returned to the ExceptionListener. A value of 0 indicates no timeout—the runtime will try indefinitely.
setReconnectInterval(int seconds)	30 seconds	Sets the interval between reconnect attempts.
setLocalStoreWaitTime(Integer seconds)	0 (indefinite wait)	Sets the wait time before messages paused by flow control are written to the local store.

Rejection Listener

The client is notified of delivery failures by a **RejectionListener** established by a method of **progress.message.jclient.Connection**:

```
setRejectionListener (RejectionListener rl);
```

The user must provide an implementation of the **RejectionListener** interface:

```
public interface RejectionListener
{ void onRejectedMessage (javax.jms.Message msg, javax.jms.JMSEException e);
}
```

The message is removed from the persistent store when **onRejectedMessage** returns.

See also [RejectionListener Semantics](#) on page 140 for additional information.

Coding Limitations

A message that is in transit when a disconnect occurs is resent when the connection is reestablished. A consumer receiving messages sent by a persistent client should be prepared to handle duplicates.

Transacted sessions and message consumers are not supported in sessions where the connection implements client persistence. The following methods return an error when the connection has local persistence:

- **Connection:** Creation of transacted sessions, `createConnectionConsumer`, `createDurableConnectionConsumer`
- **Session:** `createBrowser`, `createDurableSubscriber`, `createReceiver`, `createSubscriber`, `createTemporaryQueue`, `createTemporaryTopic`, `setMessageListener`

Note: An application can create a separate connection without persistence to use message consumers and transacted sessions.

Client persistence can be combined with fault-tolerant connections. For information about the considerations involved, see [Client Persistence and Fault-Tolerant Connections](#) on page 149.

Asynchronous Message Delivery

Asynchronous message delivery provides increased performance for delivery modes that are not explicitly asynchronous—**NON_PERSISTENT** on a security-disabled broker and **NON_PERSISTENT_ASYNC** delivery mode. This feature adds asynchronous operation to the **NON_PERSISTENT_REPLICATED** delivery mode, a delivery mode used by fault-tolerant brokers replicating nonpersistent messages from the active peer to its standby.

Asynchronous message delivery does not impact and is not applicable to **DISCARDABLE** delivery mode, or delivery within a transaction.

Asynchronous message delivery can be set in the connection factory to address the following challenges associated with asynchronous message delivery:

- **Close Behavior** — When asynchronous delivery mode is used, there may be some messages still in client buffers that have not been delivered to (or acknowledged by) the broker. This could be caused by SonicMQ flow control, TCP flow control, or just several sends followed immediately by a `close`.
- **Error propagation on failed sends** — When sending asynchronously, it is not possible to throw an exception to the caller on the send call if there is a problem with the message send. Examples of send errors are: the connection was dropped, an ACL check failed, queue or node was not found error, and message too large for a queue.
- **Number of in doubt messages** — For some applications, it is good practice to limit the number of messages that can be in-doubt, in case of an application failure or connection drop. JMS defines the number of in-doubt messages as **1** for each session.

Delivery Mode Behavior

Whether messages are delivered to the broker synchronously or asynchronously can be set on the **ConnectionFactory** by the method:

```
ConnectionFactory.setAsynchronousDeliveryMode(Integer mode)
```

or, using the constants:

```
ConnectionFactory.setAsynchronousDeliveryMode.ASYNC_DELIVERY_MODE_DEFAULT  
ConnectionFactory.setAsynchronousDeliveryMode.ASYNC_DELIVERY_MODE_ENABLED  
ConnectionFactory.setAsynchronousDeliveryMode.ASYNC_DELIVERY_MODE_DISABLED
```


The following table describes how the setting for asynchronous delivery mode at the **ConnectionFactory** level is handled for each non-transacted **MessageProducer** or **MessageDeliveryMode**:

Delivery Mode	Asynchronous Delivery Mode Setting		
	DISABLED	ENABLED	DEFAULT
PERSISTENT	Synchronous	Allowed ¹	Synchronous
NON_PERSISTENT_REPLICATED	Synchronous	Asynchronous	Synchronous
NON_PERSISTENT	Synchronous	Asynchronous	Q: Synchronous ² T: Asynchronous ^{2,3}
NON_PERSISTENT_SYNC ⁴	Synchronous	Synchronous	Synchronous
NON_PERSISTENT_ASYNC ⁴	Asynchronous	Asynchronous	Asynchronous
DISCARDABLE	Asynchronous	Asynchronous	Asynchronous

Note:

1. **PERSISTENT** Delivery is allowed on an **ASYNC_DELIVERY_MODE_ENABLED** connection.
2. Queue messages are sent synchronously, Topic messages are sent asynchronously.
3. When security is enabled on the broker, Topic messages are sent synchronously.
- 4 Specified **DeliveryMode** is deprecated.

Reliability of Produced Messages

The meaning of successful delivery of a message to the broker depends on the delivery mode with which the message is sent. The available delivery modes are as follows:

- **PERSISTENT** — The message has been received by the broker and will be delivered to each durable subscriber even if the broker crashes or fails over. If the producer is fault tolerant, message doubt can be resolved in transient connection failures or failovers.
- **NON_PERSISTENT_REPLICATED** — The message has been received by the broker and additionally replicated to the broker's backup. Messages will be sent to fault tolerant subscribers and durable subscribers unless the active broker is brought down from any state other than **ACTIVE**. If the producer is fault tolerant, message doubt can be resolved in transient connection failures or failovers.
- **NON_PERSISTENT** — The message has been received by the broker, but is not guaranteed to survive if the broker goes down or fails over to a backup. If the producer is fault tolerant messages can be lost in a transient connection failure or failover. The broker can be configured to instruct clients to internally upgrade messages produced with **NON_PERSISTENT** delivery mode to **NON_PERSISTENT_REPLICATED**.
- **DISCARDABLE** — This delivery mode can only be used by **TopicPublishers** and non-transacted sessions. The client and broker make a best effort to deliver the message to receivers, but the message will be discarded if it would trigger flow control or otherwise block the publisher. **DISCARDABLE** messages will not survive broker failure or client application/connection failure. **DISCARDABLE** messages are not fault tolerant.

Synchronous Message Reliability

For synchronously produced messages, the reliability guarantees described above are met when the send or publish call returns without an error. Otherwise, until the send call returns, message delivery to the broker is in doubt. Because sessions are single threaded at most one message is ever in doubt at any time per session.

Asynchronous Message Reliability

For asynchronously produced messages, the reliability guarantee is not met until the session is closed. If **close** returns successfully, then all asynchronously produced messages have been delivered to (or rejected by) the broker; otherwise, all asynchronously produced messages are in doubt. See [Close Behavior](#) on page 139 for more details.

Setting a connection's delivery doubt window can limit the number of asynchronously produced messages that are in doubt at any time. See [Close Behavior](#) on page 139 for more information.

Setting a RejectionListener on the connection is critical for ensuring that asynchronously produced messages have been accepted by the broker. The broker may reject a message sent to it for any of the reasons that would cause a synchronous send to throw a JMSEException. See [RejectionListener Semantics](#) on page 140 for more information.

Ordering of Asynchronously Produced Messages

Messages sent asynchronously follow the same ordering guarantees outlined in [Message Ordering and Reliability](#). Messages that are of the same **DeliveryMode**, **Destination** and **JMSPriority** define an ordered stream of messages to the broker. Using asynchronous delivery can complicate ordering if a particular message in a stream is rejected by the broker, as subsequent messages in the stream that do not produce an error are successfully delivered.

Delivery Doubt Window

Delivery of asynchronously produced messages is in doubt until the session on which they are produced is successfully closed. A connection's delivery doubt window can be set by the method:

```
ConectionFactory.setDeliveryDoubtWindow(Integer numMessages)
```

Specifying a value of 0 indicates that there is no explicit limit on the number of asynchronously produced messages at a given time.

Setting a positive integer value n for the number of messages in the delivery doubt window means that the send call will block when n number of messages have not been reliably delivered to the broker. Therefore, setting this value limits the number of messages that can be lost in a failure.

Failure conditions when fault tolerant brokers are not in use include an application crash, connection failure, broker failure, or an unsuccessful attempt to close the session.

Brokers licensed for fault tolerance that are replicating messages under **PERSISTENT** and **NON_PERSISTENT_REPLICATED** delivery mode do not experience message loss due to a connection or broker failure.

Close Behavior

Asynchronously produced messages are in doubt until a session is successfully closed. A successful close of a session is accomplished by either **session.close()** or **connection.close()** returning without throwing an exception. Upon successful close, any message not reported on the connection's **RejectionListener** has been successfully delivered to the broker. If **close** does not complete successfully, then the delivery status of any message not reported on the connection's **RejectionListener** is in doubt. In other words, if **close** generates an error, it is not guaranteed that all indoubt messages are reported on the **RejectionListener**. This is because some JMS delivery modes will not be acknowledged by the broker, and, as such, their acknowledgement is implicitly achieved by the synchronous **close** call. If the **close** call throws an exception, then some messages might not be reported on the **RejectionListener**.

Close Timeout

When messages are produced asynchronously, it is possible that some messages are pending delivery when **close** is called. These messages may be in transit to the broker, in a client queue, or being processed by the broker. If the producer is flow controlled or the client has a large backlog of undelivered messages, it is possible that **close** could take a substantial amount of time. For applications that are unwilling to wait for asynchronous message delivery to complete and are capable of handling message loss, a **close** timeout can be configured that specifies how long to wait for undelivered or in-doubt messages to reach the broker.

```
ConnectionFactory.setDeliveryCloseTimeout(Long timeout)
```

If the **close** timeout value is **-1**, **close** will block until message delivery is either complete or there is an exception in **close**. If the connection's **close** timeout value is greater than or equal to **0**, any asynchronously produced messages for which delivery guarantees cannot be met within the timeout are reported on the connection's **RejectionListener** with a **JMSAsyncDeliveryException** and an error code of **ERR_JMS_DELIVERY_TIMEOUT_ON_CLOSE**. The delivery status of such a message is **in doubt**. Delivery timeout will not cause an exception to be thrown from **close**. Therefore, in the absence of another exception being thrown, any message not reported on the **RejectionListener** were successfully delivered. In the absence of an exception on **close**, all undelivered messages are reported on the **RejectionListener**.

Although the default value for close timeout is **0**, it is recommended that you increase this value, or set it to **-1** when using asynchronous delivery.

Delivery **close** timeout also applies to synchronously produced messages. If a call to **publish()** or **send()** is blocked waiting for delivery to complete, it will throw a **JMSException** with the error code **ERR_JMS_DELIVERY_TIMEOUT_ON_CLOSE** if the timeout is reached during **close**.

Delivery **close** timeout also applies to **DISCARDABLE** messages. **DISCARDABLE** messages are written to the wire—in a best effort to deliver the messages—instead of dropping them when a **close** timeout is specified. However, if the session becomes flow controlled, **DISCARDABLE** messages are dropped.

The **close** timeout does not specify a timeout for the entire **close** operation, only the amount of time that **close** will wait for message delivery to complete.

RejectionListener Semantics

The **RejectionListener** reports asynchronously delivered messages that could not be delivered to the broker. All asynchronous delivery failures are reported using a **JMSAsyncDeliveryException**. For errors that are reported by the broker, the **JMSAsyncDeliveryException** will generally include a linked **JMSException** with the delivery error. When this is the case, **JMSAsyncDeliveryException.getErrorCode()** returns the error code of the linked exception.

When **ASYNC_DELIVERY_MODE_ENABLED** is not set, delivery failures are not reported on the **RejectionListener** to avoid breaking existing applications that use the **RejectionListener**. Instead these messages are silently dropped (preserving the existing behavior).

Because the **RejectionListener** is single threaded, the connection serializes its execution.

If a **RejectionListener** results in an unchecked exception being thrown it will be caught, and the **RejectionListener** will not be called again. It is good practice to ensure that all unchecked exceptions are handled by a **RejectionListener** implementation.

Because **session.close()** cannot complete until all **RejectionListener** calls have completed, **RejectionListener** implementations should take care not to perform any operations that could block for an extended period of time. A **RejectionListener** must never call a JMS **close** method. This restriction is in place because **close** itself cannot complete prior to all **RejectionListener** calls returning.

Once a session is closed, the **RejectionListener** will no longer be called for any messages that were produced by any of its **MessageProducers**, regardless of whether or not **close** returned successfully.

A delivery failure for a **DISCARDABLE** message is never reported on the **RejectionListener**.

The **RejectionListener** does not guarantee that errors are reported in any particular order.

The **RejectionListener** should be set prior to sending any asynchronous messages. If the **RejectionListener** is set after sending an asynchronous message then applications can't count on errors being reported. You can change the **RejectionListener**—doing so will cause new delivery failures to be reported on the new **RejectionListener**. You could unset the **RejectionListener** by specifying a null **RejectionListener**, but it is not recommended.

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 synchronization protocol exchanges, allowing it to resynchronize 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. There are several possibilities:

- 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, which fails, the fault-tolerant connection can repeatedly try to reconnect to the broker, until it is recovered and restarted.
- If you have configured and deployed a backup broker, and the primary broker fails, the fault-tolerant connection can connect to the backup broker.

Fault-tolerant connections are designed to provide continuous operation across failures for JMS operations that are 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 *Aurea SonicMQ Deployment Guide*. For more information about configuring fault-tolerant brokers, see the *Aurea SonicMQ Configuration and Management Guide*.

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

How Fault-Tolerant Connections are Initially Established

To initially establish a fault-tolerant connection, the client runtime works through a connection URL list, which can include multiple broker URLs. You add URLs to this list by calling the **ConnectionFactory.setConnectionURLs()** method.

If your client application wants 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 will not be able to initially connect.

See [Alternate Connection Lists](#) on page 120.

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 either of two ways:

- In sequential order, starting from the beginning of the list (this is the default behavior).
- In sequential order, 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 ConnectionFactory
cf.setSequential(false);
```

```
//primary and backup brokers paired in list
cf.setConnectionURLs("B1P,B1B,B2P,B2B,B3P,B3B,B4P,B4B");
```

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);
//primary and backup brokers paired in list
cf.setConnectionURLs("B1P,B1B,B2P,B2B,B3P,B3B,B4P,B4B");
```

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

```
cf.setSequential(true);
//primary brokers listed before backup brokers
cf.setConnectionURLs("B1P,B2P,B3P,B4P,B1B,B2B,B3B,B4B");
```

However, in this snippet, the primary brokers are listed before their corresponding backup brokers. This approach would be appropriate, for example, if the backup brokers were on slower machines than the primary brokers.

ConnectionFactory Methods for Fault-Tolerance

The **ConnectionFactory** class has several methods related to fault tolerant connections, as shown in the following table. The usage of these methods is described in more detail in the sections following the table.

Table 6: ConnectionFactory Methods for Fault-Tolerance

Method Signature	Description
Long getClientTransactionBufferSize()	Gets the client transaction buffer size.
Boolean getFaultTolerant()	Indicates whether new Connections will be fault tolerant.
Integer getFaultTolerantReconnectTimeout()	Gets the fault tolerant reconnect timeout.
Integer getInitialConnectTimeout()	Returns the initial connect timeout.
setClientTransactionBufferSize(Long size)	Sets the client transaction buffer size.
setFaultTolerant(Boolean faultTolerant)	Enables and disables fault tolerance for new connections.
setFaultTolerantReconnectTimeout(Integer seconds)	Sets the fault tolerant reconnect timeout.
setInitialConnectTimeout(Integer seconds)	Sets the initial connect timeout.

Enabling Fault-Tolerant Connections

By default, a **ConnectionFactory** creates standard connections, not fault-tolerant ones. If you want to create a fault-tolerant connection, you must first call the following method:

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 **progress.message.jclient.Connection.isFaultTolerant()** method.

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're using broker replication. If the client runtime is able to resume the connection with either broker, it must make sure that its transaction state is synchronized with the broker's transaction state.

The broker, for performance reasons, 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 tuning parameter, **Transactions: Buffer Size**, 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 transaction buffer size. This is done 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: BufferSize** 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, JMS 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 JMS Session can have at most one transaction active. For global transactions, every active XA transaction branch is considered an active transaction.

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

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

public Long 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** — Indicates how long the client runtime tries to establish an initial connection to the broker
- **Fault tolerant reconnect timeout** — Indicates how long the client runtime tries to resume a fault tolerant connection after a problem is detected

To set the initial connect timeout, call the following method:

ConnectionFactory.setInitialConnectTimeout(Integer timeout)

The default timeout is 30 seconds.

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 141).

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 will 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.

Note: The **InitialConnectTimeout** setting interacts with the setting described in [Setting a Socket Connect Timeout](#) on page 121, and—for fault tolerant connections—the operating systems settings discussed in the “Tuning TCP to Optimize CAA Failover” in the *SonicMQ Performance Tuning Guide*. The socket connect timeout should allow for an attempt at every listed URL. For example, where a URL list contains six URLs, the default setting for the **InitialConnectTimeout** of 30 seconds would require that the **SocketConnectTimeout** value be set to 5 seconds. The tuning of the operating system for fault tolerant failover assures that the OS does not add unintended delays.

When you call the **setInitialConnectTimeout()** method, valid values are as follows:

- **Positive non-zero value** — Specifies a timeout; the client runtime will abandon further connection attempts if the timeout is exceeded.
- **Zero (0)** — Specifies no timeout; the client runtime will try indefinitely.
- **Negative one (-1)** — Specifies that each URL is tried one time only; the client runtime will try each URL sequentially one at a time until a successful connection is made or until all URLs have been 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 will be thrown.

To set the fault tolerant reconnect timeout, call the following method:

ConnectionFactory.setFaultTolerantReconnectTimeout(Integer timeout)

The default timeout is 60 seconds.

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 will; otherwise, it will try to reconnect to a standby broker (if you are using broker replication).

When the client runtime successfully establishes a fault tolerant connection with a broker, the broker sends a list of URLs to the client runtime to be used for the purpose of reconnection. If replicated, the broker also sends a list of standby broker URLs for the purpose of reconnection. After a connection is established, you can see the values in these lists by calling the following methods:

- `progress.message.jclient.Connection.getBrokerReconnectURLs()`
- `progress.message.jclient.Connection.getStandbyBrokerReconnectURLs()`

When you call the `setFaultTolerantReconnectTimeout()` method, valid values are as follows:

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

If the client is using the client persistence feature and the client runtime fails to reconnect, the connection will go offline. For more information, see [Client Persistence and Fault-Tolerant Connections](#) on page 149.

If the client is **not** using the client persistence feature and 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`.

The client's ability to reconnect is also influenced by the advanced broker property **Client Reconnect Timeout**. The default timeout is **600** seconds—10 minutes. This property limits the overall length of time the broker will maintain 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 value set in **Client Reconnect Timeout**.

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.

Connection Methods for Fault-Tolerance

The `progress.message.jclient.Connection` class has several methods related to fault tolerant connections, as shown in the following table. The usage of these methods is described in more detail in the sections following the table.

Table 7: Connection Methods for Fault-Tolerance

Method Signature	Description
<code>int getConnectionState()</code>	Returns the current connection state.
<code>ConnectionStateChangeListener getConnectionStateChangeListener()</code>	Returns the current <code>ConnectionStateChangeListener</code> .
<code>setConnectionStateChangeListener(ConnectionStateChangeListener listener)</code>	Sets the current <code>ConnectionStateChangeListener</code> .
<code>String getBrokerURL()</code>	Returns the URL of the currently connected broker.
<code>String[] getBrokerReconnectURLs()</code>	Returns a <code>String</code> array containing all of the URLs that the client runtime can use to try to resume a connection to the connected broker.

Method Signature	Description
<code>String[] getBrokerStandbyReconnectURLs()</code>	Returns a String array containing all of the URLs that the client runtime can use to try to resume a connection to a backup broker.
<code>boolean isFaultTolerant()</code>	Returns true if the connection is fault tolerant.

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 exceptional condition was raised. If the client application was in the middle of a synchronous call, the exception would be thrown by the invoked method. If the exception occurred asynchronously, the client runtime would pass 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 it is 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 it would 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):

- **progress.message.jclient.Constants.ACTIVE** — The connection is active.
- **progress.message.jclient.Constants.RECONNECTING** — The connection is unavailable, but the client runtime is trying to resume the connection.
- **progress.message.jclient.Constants.FAILED** — The client runtime has tried to reconnect and failed.
- **progress.message.jclient.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 will never get 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 JMS operations related to the connection, except for calling the following informational methods:

- **progress.message.jclient.Connection.getConnectionState()**
- **progress.message.jclient.Connection.getBrokerURL()**
- **progress.message.jclient.Connection.getBrokerReconnectURLs()**
- **progress.message.jclient.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

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 all these cases, a client application can determine which broker it is currently connected to by calling the **progress.message.jclient.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 may 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 **progress.message.jclient.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 **progress.message.jclient.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 and reconnect to the current broker, in the event of connection failure (transient or other). The broker reconnect URLs allows 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 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 to 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 will be supported with fault-tolerant connections is **PERSISTENT** to **PERSISTENT**.

PERSISTENT to **NON_PERSISTENT**, and vice-versa, will throw an **IllegalStateException** when attempted on a fault-tolerant connection.

Forward and Reverse Proxies

Fault-tolerant connections will work through forward proxy servers.

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

Client Persistence and Fault-Tolerant Connections

Fault-tolerant connections and client persistence can be used together. When you use these features together, you need to understand how the client runtime makes the transition from ordinary messaging to client persistence, and vice versa.

You need to understand the purpose of the settings in the following table and how they affect client behavior.

Table 8: Timeout Settings

Setting	progress.message.jclient.ConnectionFactory Method
Initial connect timeout	setInitialConnectTimeout()
Fault-tolerant reconnect timeout	setFaultTolerantReconnectTimeout()
Reconnect timeout	setReconnectTimeout()
Reconnect interval	setReconnectInterval()

When the client runtime initially establishes a fault-tolerant connection, it checks the value of the initial connect timeout, set with the **setInitialConnectTimeout()** method. This method determines how long the client runtime tries to establish an initial fault-tolerant connection.

After the fault-tolerant connection is successfully established, it will continue to operate normally until a problem occurs with the network or broker. If a problem occurs, the client runtime will try to resume the connection. The **setFaultTolerantReconnectTimeout()** method determines how long the client runtime attempts to resume the connection.

While the client runtime tries to resume the fault-tolerant connection, the persistent client is still online. However, once the fault-tolerant reconnect timeout expires, the persistent client goes offline, and JMS message sends are saved to the client's local disk.

While offline, the persistent client runtime internally attempts to reconnect. This process is controlled by two persistent client settings: reconnect interval and reconnect timeout. The **setReconnectInterval()** method determines the interval between reconnect attempts. The **setReconnectTimeout()** method determines how long the client runtime tries to reconnect before returning an exception to the application; this method effectively puts a cap on how long the persistent client is willing to operate offline.

The client persistence feature is essentially indifferent to the type of connection you are using, whether standard or fault-tolerant. The only difference between a standard connection and a fault-tolerant connection is *when* the transition to client persistence takes place. If a standard connection has a problem with the broker or network, the connection is immediately dropped, and the transition to client persistence immediately follows. If a fault-tolerant connection has a problem with the broker or network, it tries to resume the connection, delaying the transition to client persistence until the fault-tolerant reconnect timeout expires.

Consider the following example. Suppose a client application wants to use the client persistence feature and combine it with fault-tolerant connections. Further suppose the client application uses the following settings:

- **Initial connect timeout** — 30 seconds
- **Fault-tolerant reconnect timeout** — 60 seconds
- **Reconnect timeout** — 360 minutes
- **Reconnect interval** — 600 seconds

When the client application initially connects to the broker, it does so within 25 seconds, so the fault-tolerant connection succeeds. The persistent client application goes online. When the client is online, JMS messages are transmitted directly to the broker. Later, the network fails, and the client runtime attempts to resume the connection, but fails to do so within 60 seconds, so the fault-tolerant reconnect timeout expires. At this time, the persistent client goes offline.

When the client is offline, JMS messages are saved on the client's local disk. The offline persistent client runtime continues to save JMS messages, but internally the runtime is attempting to reconnect to the broker. This process is controlled by two persistent client settings: reconnect interval and reconnect timeout. After every reconnect interval, the persistent client will attempt to reconnect. If the reconnect timeout is exceeded the persistent client will fail and return an exception to the application. By default, the reconnect timeout is set to 0, which means that the client runtime will continually try and connect to the broker.

Continuing this example, suppose the broker restarts after 15 minutes. Since the reconnect interval is set to 600 seconds (10 minutes), on the second reconnect attempt the client will succeed and go back online. In this case the client operates offline for a period of 20 minutes.

JMS Operation Reliability and Fault-Tolerant Connections

Reliability refers to resilience after a *broker failure*—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. The reliability of various JMS and SonicMQ-specific operations in the event of a client reconnect after a failure are as follows:

- 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 will throw a **java.util.NoSuchElementException** exception with text indicating that the browse has been terminated due to fail-over. Any attempt to call a **QueueBrowser** method after fail-over will result in a **javax.jms.IllegalStateException**. Explanatory text is provided in the exception. The following String error code will be provided:
`progress.message.jclient.ErrorCodes.BROWSER_CLOSED_DURING_RECONNECT`
- Access to read-exclusive queues (a Sonic-specific feature) may 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 will happen if another client opens a receiver to the same queue before the fault-tolerant client reconnects. In this case, normal JMS 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 acknowledgement 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 has synchronized 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 have been received.

Reconnect Conflict

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

JMS Connection Reconnect Conflict

To uniquely identify connections, the **ConnectionFactory username** and **connectID** values are used. **A non-null connectID is required**, so that only one connection with the particular **username** and **connectID** combination is permitted, and connected when failover 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 inelegantly 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 has failed and is pending reconnect. Therefore if a fault-tolerant connection has 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 will be accepted and the previous connection state will be discarded.

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.

This means that 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 **javax.jms.Session.createDurableSubscriber** are used. A client may 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

The following table describes message reliability levels for clients that reconnect to the 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 reconnecting using standard connections do not resend in-doubt messages upon reconnecting.

Asynchronous message delivery set on the **ConnectionFactory** allows for a wider delivery doubt window than possible with asynchronous message production, as the reliability guarantee is not met until the session is closed. Setting a connection's delivery doubt window can limit the number of asynchronously produced messages that are in doubt at any time. See [Asynchronous Message Delivery](#) on page 136 for more information.

Note: The only way to guarantee exactly-once delivery is to use a fault-tolerant persistent **MessageProducer** and a fault-tolerant **MessageConsumer**.

Table 9: 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

¹In 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 had received the original message. According to JMS this is not 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.²In the case of a standard connection failure, the acknowledgement for the last message may be lost. In this case the broker will redeliver the message with **JMS_REDELIVERY** set to true in accordance with the JMS Specification.³If a message consuming client reconnects using a standard connection at the same time as a fault-tolerant publisher is reconnecting, it is possible that the publisher will resend a message that had been delivered to the previously connected client. According to JMS this is not a redelivery since the message was delivered to a new session.

NON_PERSISTENT_REPLICATED Delivery Mode

SonicMQ provides the **NON_PERSISTENT_REPLICATED** delivery mode for fault-tolerant deployments. For messages sent with this delivery mode, SonicMQ will protect against message loss due to broker failures by replicating the messages to the standby broker. This feature is **Fast Forward** mode in Sonic's Continuous Availability Architecture (CAA-FF).

Note: In contrast, if the delivery mode is **NON_PERSISTENT**, SonicMQ does not replicate the messages. However, setting the advanced broker property **BROKER_FAULT_TOLERANT_PARAMETERS.FT_REPLICATE_NON_PERSISTENT** to **true**, upgrades the delivery mode of any messages sent using the **NON_PERSISTENT** delivery mode to **NON_PERSISTENT_REPLICATED** for a fault tolerant deployment. For more information on setting this and other exposed advanced broker properties, see the "Configuring MQ Brokers" chapter of the *SonicMQ Configuration and Management Guide*.

The **NON_PERSISTENT_REPLICATED** delivery mode also ensures once-and-only-once deliver to fault-tolerant subscribers (both durable and non-durable), provided that after a failure the subscriber either successfully resumes its connection at the same broker or fails over to the standby broker.

This delivery mode provides a more satisfactory level of performance for applications that do not want to use **PERSISTENT** messages. If the delivery mode is **PERSISTENT** and a durable subscriber consumes the message, SonicMQ replicates the messages to the standby broker (as with **NON_PERSISTENT_REPLICATED**) but also persists the messages to the recovery log (which requires disk I/O).

The **NON_PERSISTENT_REPLICATED** delivery mode instructs SonicMQ to protect the message in the event of the following types of failure:

- A client application submits a message to a fault-tolerant broker that is in the **ACTIVE** state and is replicating messages to a standby broker. If the active broker fails, the application fails over to the standby broker. The message is not lost. Also, the message is not redelivered to its consumers, provided that both the producer and the consumers use fault-tolerant connections.
- A client application submits a message to a fault-tolerant broker that is working in the **STANDALONE** or **ACTIVE_SYNC** replication state. Later, the messaging state of the standby broker is synchronized with the active broker. The active broker is now running in the **ACTIVE** replication state.

If the active broker, **B1**, fails, the application fails over to the configured backup broker standby broker, **B1_BU**. The message is not lost. Also, the message is not redelivered to its consumers, provided that both the producer and the consumers use fault-tolerant connections.

The standby broker **B1_BU** runs in the **STANDALONE** replication state until its peer broker, **B1**, restarts, establishes a replication connection between **B1** and **B1_BU**, and starts synchronizing its data to the active broker's data. When the brokers are fully synchronized, **B1_BU** assumes the active role and **B1** assumes the standby role.

Important: Time is of the essence — When synchronization is in process, the messages produced with the **NON_PERSISTENT_REPLICATED** delivery mode are not protected from a crash of broker **B1_BU**. That is, if broker **B1_BU** fails while running in the **STANDALONE** or **ACTIVE_SYNC** replication state, the **NON_PERSISTENT_REPLICATED** messages can be lost or redelivered (or both) when **B1_BU** is restarted, because the messages and their acknowledgements have not been persisted to the recovery log. In this scenario, applications cannot failover to broker **B1** because it does not have its messages completely synchronized with broker **B1_BU** and, therefore, it does not failover and does not accept client connections. When an active broker fails and the client connections failover to the standby broker, it is very important to recover and restart the failed broker as soon as possible. Otherwise, if the other broker also fails, message loss or duplication (or both) can occur. See the section “Recovery of a Broker” in the chapter “Broker Replication” in the *Aurea SonicMQ Deployment Guide* for detailed instructions for both recoverable interruptions and disaster recovery. When the crash of the active broker is followed by a prompt, successful restart of the failed broker, and then recovery to the protected state (one broker in the **ACTIVE** state and its peer in the **STANDBY** state), **NON_PERSISTENT_REPLICATED** messages are protected and no loss or duplication of messages occurs.

- A client application submits a message to a broker. In this type of failure, the broker does not have to be configured for replication but it does have to be licensed for fault-tolerance.

Important: Exactly-once recovery for the broker's recovery logs must be enabled on your broker. This feature is enabled by default on every broker but if you had been advised by your Sonic representative to clear the **XONCE Recovery** option on the broker's **Tuning** properties,

consult with your Sonic representative to determine whether the setting can be selected (set to **true**) at this time.

- A client application experiences a transient network failure. In this type of failure, the SonicMQ client runtime successfully resumes its connection at the same broker. The message is not lost. Also, the message is not redelivered to its consumers, provided that both the producer and the consumers use fault-tolerant connections.

Failures That Cause Message Loss or Duplication

There are severe circumstances that can result in message loss or redelivery/duplication of **NON_PERSISTENT_REPLICATED** messages:

- If a broker is restarted when it is running without a standby or the broker's replication state is not **ACTIVE**.
- If the broker is running in the **ACTIVE** state and both the active broker and the standby broker crash before the applications can failover.

Lost Messages

In these cases, **NON_PERSISTENT_REPLICATED** messages might be lost once the brokers are restarted even if client applications resume their connections and sessions without receiving an exception. Some messages can be lost because they have not been written to the recovery log.

Redelivered/Duplicated Messages

In these cases, **NON_PERSISTENT_REPLICATED** messages might be redelivered. This might happen because when a consumer acknowledges a **NON_PERSISTENT_REPLICATED** message, the broker does not record the acknowledgement in its recovery log.

Setting the Default Delivery Mode for a Message Producer

An application can use the **setDeliveryMode(int deliveryMode)** method in the **MessageProducer** class to set the default delivery mode to **NON_PERSISTENT_REPLICATED**.

The default delivery mode is used when an application calls a variation of the **send()** or **publish()** methods that do not have the delivery mode as one of their arguments. These methods are defined in the **MessageProducer**, **QueueSender** and **TopicPublisher** classes.

Note: Enabling asynchronous message delivery on the **ConnectionFactory** interprets **NON_PERSISTENT_REPLICATED** delivery mode as an asynchronous delivery yet supported by a specified indoubt window and timeout and management of delivery through the close of the session. See [Asynchronous Message Delivery](#) on page 136 for more information.

Overriding the Default Delivery Mode on a Message

An application can request the **NON_PERSISTENT_REPLICATED** delivery mode explicitly in several signatures of **send()** and **publish()** methods, thereby overriding the delivery mode of the message producer for the message that is being sent or published.

Note: While an application can pass the **NON_PERSISTENT_REPLICATED** delivery mode to the **setJMSDeliveryMode()** method in the **javax.jms.Message** interface, the value set by this method is used only to return it when the application calls the **getJMSDeliveryMode()** method.

You can use this setting to restate the selected delivery mode into the message so that it can be retrieved by the consumer as for informational use.

Redelivery of **NON_PERSISTENT_REPLICATED** Messages

If a message consumer uses a fault-tolerant connection and does not specify the **DUPS_OK** acknowledgement mode, SonicMQ guarantees once-and-only-once delivery for the **NON_PERSISTENT_REPLICATED** messages in presence of the failures described in [Nondurable Subscribers of NON_PERSISTENT_REPLICATED Messages](#) on page 156. This means that no message are delivered to the consumer more than once.

If the consumer uses the **DUPS_OK** acknowledgement mode, **NON_PERSISTENT_REPLICATED** messages can be redelivered to the consumer after a failure.

Regardless of the acknowledgement mode, **NON_PERSISTENT_REPLICATED** messages can be redelivered after any of the failures described in [Failures That Cause Message Loss or Duplication](#) on page 155.

Nondurable Subscribers of **NON_PERSISTENT_REPLICATED** Messages

In the case of topic messages, once-and-only-once delivery is guaranteed for **NON_PERSISTENT_REPLICATED** messages even if the subscriber is non-durable. That is different from the behavior of the **PERSISTENT** messages received by non-durable subscribers; those messages can be redelivered after a failure.

SonicMQ processes **NON_PERSISTENT_REPLICATED** messages sent to a non-durable subscriber in a manner that is similar to how it processes **PERSISTENT** messages sent to a durable subscriber. (Note that **PERSISTENT** messages to a non-durable subscriber are treated as **NON_PERSISTENT**—they are neither replicated nor logged.)

The main differences are as follows:

- Unlike a **PERSISTENT** message, a **NON_PERSISTENT_REPLICATED** message is not written to the recovery log.
- When the subscriber application closes the subscriber, the subscription is deleted from the broker and remaining unprocessed messages are dropped.

Clients Resuming Fault Tolerant JMS Sessions on an Unclustered Broker

When fault tolerant connections are used by message publishers sending **NON_PERSISTENT_REPLICATED** messages that are received by non-durable fault-tolerant subscribers, then once-and-only-once delivery of messages is guaranteed. But action on a fault tolerant connection might not be broker interruptions or failover, it might be when the subscriber experiences a temporary network failure. In that case, the client resumes its JMS session at the same broker without message loss or duplication.

If the subscriber does not resume its session before the reconnect timeout expires, the subscription is deleted, the unprocessed messages are dropped.

Clients with Fault Tolerant JMS Sessions on Clustered Brokers

When brokers are clustered, it could occur that the message publisher has a fault tolerant connection on one cluster member, **ClusterA_B1**, while the message consumer has a fault tolerant connection to a another cluster member, **ClusterA_B2**. If the messages are published with the **NON_PERSISTENT_REPLICATED** delivery mode, then once-and-only-once delivery is guaranteed. That's the same as the behavior on an unclustered broker.

If broker **ClusterA_B1** loses its connection to broker **ClusterA_B2**, the new **NON_PERSISTENT_REPLICATED** messages are retained at broker **ClusterA_B1** as described in the previous section. This does not present a problem if the connection loss is caused by a temporary network failure, because broker **ClusterA_B1** only needs to retain the new messages for a short period of time.

If broker **ClusterA_B2** is a fault-tolerant replicated broker pair, when **ClusterA_B2** fails over to its backup (**ClusterA_B2_BU**), **ClusterA_B1**'s interbroker connection and the subscriber's client connection failover to **ClusterA_B2_BU**. Therefore, **ClusterA_B1** does not need to retain the new messages for a long time.

In situations where **ClusterA_B1** cannot reconnect to **ClusterA_B2** (or its peer) for a long time, retention of the new messages published to the subscriber's topic might present a problem because writing messages to the database affects **ClusterA_B1**'s performance. It is also possible that the database of broker **ClusterA_B1** becomes full.

The salient difference between the cluster case and the single broker case is that in the case of a single broker there is a reconnect timeout that prevents the broker from retaining new messages for the disconnected subscriber for a long time. In the cluster case, there is no such timeout.

Therefore broker **ClusterA_B1** retains the subscription and retains new **NON_PERSISTENT_REPLICATED** messages published to the subscriber's topic, until it can reconnect to a broker in the **ClusterA_B2** replicated pair. At that point, if the subscriber application has already closed the subscriber, the subscription and its messages are deleted. If the subscriber is still connected at **ClusterA_B2**, the retained messages are delivered to the subscriber.

If you know that a broker member and its peer will be unavailable for a long time, remove that broker from the cluster. That deletes the subscription at **ClusterA_B1**, so that **ClusterA_B1** stops retaining new messages published to the subscriber's topic.

Broker Storage of NON_PERSISTENT_REPLICATED Messages

Even though **NON_PERSISTENT_REPLICATED** messages are not written to the recovery log, they can still incur disk I/O overhead if the broker writes them into its persistent storage.

The broker can write a **NON_PERSISTENT_REPLICATED** message to persistent storage in one of the following situations:

- A **NON_PERSISTENT_REPLICATED** message is a queue message and the in-memory save extent of its destination queue is full. The broker writes the message to the database part of the queue. Note that this does not happen if the max size of the queue is less or equal to the size of its save extent.
- A **NON_PERSISTENT_REPLICATED** message is a topic message that needs to be delivered to one or more durable subscribers and some of those subscribers are disconnected. The broker will store the message in the database. The message is read back from the broker's persistent storage when the subscriber connects back.

To avoid that, you can either use non-durable subscription or configure their durable subscriptions with a very short expiration period so that a subscription is deleted after the subscriber gets disconnected. That means that all messages published after the subscription has expired and before the subscriber reconnected are lost.

Note that there is a difference between resuming a fault-tolerant connection and disconnecting. The latter means that either the application disconnected on its own or the pending reconnect interval has expired before SonicMQ client runtime attempted to resume the lost connection.

- A **NON_PERSISTENT_REPLICATED** message is a topic message and the flow-to-disk feature is in effect. If a subscriber (durable or non-durable) falls behind and its messages fill up one of the subscriber's in-memory buffers in the broker (see the *SonicMQ Performance Tuning Guide*), subsequent messages are stored in the broker's persistent storage. They are read back from persistent storage when the subscriber processes and acknowledges enough messages to make room for more.
- A **NON_PERSISTENT_REPLICATED** message is a topic message and a fault-tolerant subscriber of the message is placed in the **PENDING_RECONNECT** state by the broker after a failure such as a crash of the active broker or a network failure on the subscriber's connection. The subscriber is in the **PENDING_RECONNECT** state at the broker until the subscriber's application resumes its connection to the broker. While the subscriber is in the **PENDING_RECONNECT** state, new **NON_PERSISTENT_REPLICATED** messages are stored in an in-memory buffer at the broker (each subscriber has one such buffer at the broker). Once that buffer becomes full, subsequent **NON_PERSISTENT_REPLICATED** messages are stored in the broker's persistent storage, and the messages are read back from the database when the subscriber resumes its connection. This behavior is the same for both durable and non-durable subscribers.

Effect of Broker Restart on NON_PERSISTENT_REPLICATED Messages

Topic and queue messages in the **NON_PERSISTENT_REPLICATED** delivery mode stored in the broker's persistent storage are processed differently when the broker is restarted:

- **Queues** — The **NON_PERSISTENT_REPLICATED** queue messages are deleted from the broker's persistent storage during broker restart.
- **Topics for Non-Durable Subscribers** — The **NON_PERSISTENT_REPLICATED** topic messages that were stored in the broker's persistent storage for non-durable subscribers either because of the flow-to-disk feature or because the subscriber's Pending Reconnect buffer became full are also deleted from the database during broker restart.
- **Topics for Durable Subscribers** — The **NON_PERSISTENT_REPLICATED** topic messages stored in the database for durable subscribers are not deleted from the database unless the subscription expires. After the broker restart, the messages are delivered to the subscriber.

NON_PERSISTENT_REPLICATED Messages in Transactions

A transaction might have producer overrides of the delivery mode such that the transaction under construction has a mixture of **NON_PERSISTENT_REPLICATED**, **PERSISTENT** and **NON_PERSISTENT** messages. When fault tolerant brokers and fault tolerant client connections are not being used, a broker failure loses the transaction in process and the client session is rolled back.

When fault tolerant brokers and fault tolerant client connections are in use, transactional behavior on a restart of a standalone broker or failover in a replicated broker pair depends on whether or not the broker had to write the transaction to a file because the transaction's in-memory buffer was filled:

- If the transaction did not fill up its transaction buffer, each message in the transaction is treated according to its delivery mode meaning that in certain situations, part of the transaction can be lost, depending on the broker configuration and state:
 - When a non-fault tolerant broker, a standalone fault tolerant broker, or both brokers in a fault tolerant pair simultaneously have to restart, only **PERSISTENT** messages will be available.
 - When a broker in a fault tolerant pair fails over to its standby, the **PERSISTENT** messages and **NON_PERSISTENT_REPLICATED** messages will be available.
- If the transaction buffer becomes full, all messages in the transaction are kept together and all of them are replicated; therefore, no messages are lost once the application has committed the transaction, depending on the broker configuration and state:
 - When a non-fault tolerant broker, a standalone fault tolerant broker, or both brokers in a fault tolerant pair simultaneously have to restart, only the **PERSISTENT** messages will be available.
 - When a broker in a fault tolerant pair fails over to its standby, the complete transaction buffer (as well as any messages from committed transactions that were **PERSISTENT** or **NON_PERSISTENT_REPLICATED**) will be available.

Using **NON_PERSISTENT_REPLICATED** in **acknowledgeAndForward**

The **acknowledgeAndForward** feature for queue messages lets an application atomically acknowledge a received message and forward it to another queue in one application call.

The application can generally request that the received message is forwarded to another queue using a delivery mode that is different from the delivery mode that was used when the message was originally produced.

The valid transitions of delivery mode for non-fault tolerant **acknowledgeAndForward** are: persistent to persistent, persistent to non-persistent, non-persistent to persistent and non-persistent to non-persistent.

However, when the forwarding application uses a fault-tolerant client connection, change is not permitted. The original delivery mode must be used as the forwarding delivery mode. The valid modes are:

- **PERSISTENT > PERSISTENT**
- **NON_PERSISTENT > NON_PERSISTENT**
- **NON_PERSISTENT_REPLICATED > NON_PERSISTENT_REPLICATED**

Using NON_PERSISTENT_REPLICATED Delivery Mode on Non-Fault Tolerant Connections

If the **NON_PERSISTENT_REPLICATED** delivery mode is used on a non-fault-tolerant connection, the SonicMQ client runtime code cannot guarantee once-and-only-once delivery. Specifically, if the message producer receives an exception as a result of some failure while trying to send or publish a **NON_PERSISTENT_REPLICATED** message, there is an uncertainty regarding whether or not the broker received the message. If the application reconnects to the broker and resubmits the message, the consumer(s) of the message may receive it twice (the **JMSUndelivered** flag will be set to **false**).

This is a specified behavior for the **PERSISTENT** messages in the JMS standard.

The above behavior also applies when where an application attempts to create a fault-tolerant connection to a broker that is not licensed for fault tolerance and then uses the **NON_PERSISTENT_REPLICATED** delivery mode.

If **NON_PERSISTENT_REPLICATED** messages are being delivered to a non-fault-tolerant consumer and a failure takes place, the consumer has to manually re-connect to the broker. In this situation, there is an uncertainty regarding the last message that was consumed prior to the failure. That message may be redelivered by SonicMQ to the consumer (the **JMSRedelivered** flag is set to **true**). This is a specified behavior for the **PERSISTENT** messages in the JMS standard.

Modifying the Chat Example for Fault-Tolerance

This section describes how to modify the **Chat** sample provided with SonicMQ to use fault-tolerant connections.

To modify the Chat sample:

1. Create a directory `MQ_install_root/samples/TopicPubSub/Chat/ChatFT`.
2. Create a copy of the file `MQ_install_root/samples/TopicPubSub/Chat/Chat.java` and paste it in the directory you just created.
3. Set the file to be write enabled.
4. Open the copied file **Chat.java** in a text editor or Java IDE.
5. In the body of the **chatter()** method, replace the first **try** block with the following:

```
try { javax.jms.ConnectionFactory factory; factory = (new
progress.message.jclient.ConnectionFactory (broker)); // Tell the ConnectionFactory to create
a fault-tolerant connection ((progress.message.jclient.ConnectionFactory)factory).
setFaultTolerant(new Boolean(true)); // Increase the default connect timeout to 90 seconds
((progress.message.jclient.ConnectionFactory)factory). setInitialConnectTimeout(new
Integer(90)); // If the connection fails, keep retrying the connection indefinitely
((progress.message.jclient.ConnectionFactory)factory). setFaultTolerantReconnectTimeout(new
Integer(0)); connect = factory.createConnection (username, password); // Set the fault-tolerant
connection's ConnectionStateChangeListener ((progress.message.jclient.Connection)connect).
setConnectionStateChangeListener(new ConnectionStateMonitor()); pubSession =
connect.createSession(false,javax.jms.Session.AUTO_ACKNOWLEDGE); subSession =
connect.createSession(false,javax.jms.Session.AUTO_ACKNOWLEDGE); }
```

6. Near the end of the file, after the **printUsage()** method body, insert the following internal class definition:

```
class ConnectionStateMonitor
implements progress.message.jclient.ConnectionStateChangeListener
{
    public void connectionStateChanged(int status)
```



```

{
System.out.println("+++++++\n");
// Check status and write appropriate message to the console
switch (status)
{
case progress.message.jclient.Constants.RECONNECTING:
System.out.println("SYSTEM: Connection is inactive. " +
"Trying to reconnect. Please wait."); break;
case progress.message.jclient.Constants.ACTIVE:
System.out.println("SYSTEM: Connection is active" +
" and operating normally."); break;
case progress.message.jclient.Constants.FAILED:
System.out.println("SYSTEM: Connection has failed." +
" Cannot reconnect."); break;
case progress.message.jclient.Constants.CLOSED:
System.out.println("SYSTEM: Connection is closed.");
}
// Write the reconnect and standby URLs to the console
String[] brokerURLs = ((progress.message.jclient.Connection)connect).
getBrokerReconnectURLs();
String[] standbyURLs = ((progress.message.jclient.Connection)connect).
getStandbyBrokerReconnectURLs();
if (brokerURLs == null)
System.out.println("SYSTEM: No broker reconnect URLs provided.");
if (brokerURLs != null){
System.out.println("SYSTEM: The broker reconnect URLs are as follows:");
for ( int i = 0; i < brokerURLs.length; ++i ){
System.out.println("Reconnect URL[" + i + "] is " + brokerURLs[i]);
}
}
if (standbyURLs == null)
System.out.println("SYSTEM: No standby broker URLs provided.");
if (standbyURLs != null && standbyURLs.length > 0){
System.out.println("SYSTEM: The standby broker URLs are as follows:");
for ( int i = 0; i < standbyURLs.length; ++i ){
System.out.println("Standby URL[" + i + "] is " + standbyURLs[i]);
}
}
}
}
}

```

7. Save the modified file.

Compiling the edited sample

To compile the edited sample:

1. Open a command line console window, change directory to the `ChatFT` directory.
2. Enter `..\..\..\SonicMQ` to run the script file that sets up the SonicMQ variables and environment.
3. Locate or install a Java SDK and the compiler, **javac.exe**.
4. Enter the path to the Java compiler, the SonicMQ classpath and the file name, in a form similar to the following: `c:\jdk\bin\javac -classpath "%SONICMQ_CLASSPATH%" Chat.java`
5. Resolve any compile time errors.

Running the Modified Chat Example

Now that you have 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, do the following:

1. Make sure the broker is running. If the broker is not running, start it. Select:

Start > All Programs > Aurea > Sonic 2015 > Start DomainManager

2. In a console window at the **ChatFT** directory, enter: `..\..\..\SonicMQ Chat -b localhost:2506 -u SALES`

This step starts a JMS client that uses a fault-tolerant connection.

3. Open another command line console window, change directory to the **Chat** directory and enter: `..\..\SonicMQ Chat -b localhost:2506 -u MARKETING`

This step starts a JMS client that uses a standard connection.

Both JMS clients will 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 will continue to try and reconnect until you explicitly shutdown 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**.

The following exception is displayed:

```
javax.jms.IllegalStateException: The session is closed.
at progress.message.jimpl.Session.GsB_(Unknown Source)
at progress.message.jimpl.Session.createTextMessage(Unknown Source)
at Chat.chatter(Chat.java:94)
at Chat.main(Chat.java:225)
```

This occurs because the session was closed when the standard connection to the broker was closed.

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

Notice that the client application appears to block. This behavior occurs because all client operations are suspended when the connection is unavailable. Also notice that no exception is displayed.

- Restart the broker by selecting:

Start > Programs > Aurea > Sonic 2013 > Start DomainManager

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 have 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 need to stop or start connections in order 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 will 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 will wait until all the message listeners have returned before it returns. MessageConsumers that are active 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 get 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.

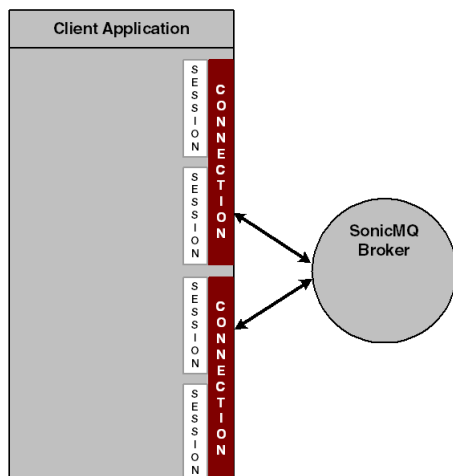
The message objects can be used in a closed connection with the exception of the message's acknowledge methods.

See [SonicMQ Client Sessions](#) for information about coding connections and sessions and handling exceptions on connections.

Using Multiple Connections

Sometimes it may 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. The following figure shows two connections to a SonicMQ broker, each with two sessions.

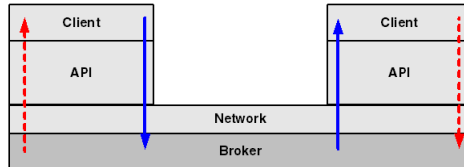
Figure 44: Multiple Connections in a Client Application



Communication Layer

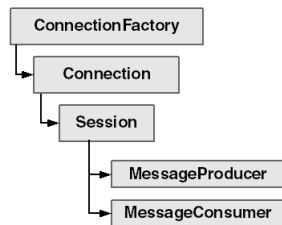
The SonicMQ broker works in concert with the network layer to provide asynchronous message communications between client applications. As shown in the following figure, a client can send and receive messages through the SonicMQ API and interfaces to communicate on network connection to a broker. Messages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

Figure 45: Client-Broker-Client Communications



The connection layer, as shown in the following figure, involves getting a **ConnectionFactory**, then creating a **Connection**, and finally creating a **Session**. A **Session** holds **MessageProducer** and **MessageConsumer** objects.

Figure 46: 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 will get 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 could be any number (between zero and many) of consumers for a given topic who will each receive the one message that was sent. In addition, a consumer can be a durable subscriber, and SonicMQ will save messages until the subscriber reconnects. If no consumers express an interest in a message topic, the message is discarded.

SonicMQ Client Sessions

This chapter explains the programming concepts and actions required to establish and maintain SonicMQ client sessions.

For details, see the following topics:

- [Overview of Client Sessions](#)
- [Session Objects](#)
- [Flow Control](#)
- [Flow to Disk](#)
- [Send Timeout](#)
- [Using Sessions and Consumers](#)
- [JMS Messaging Domains](#)
- [Integration with Application Servers](#)

Overview of Client Sessions

The SonicMQ Java client provides a lightweight platform that can access the messaging features provided by the SonicMQ brokers. In the JMS programming model, a programmer creates JMS connections that establish the application's identity and specify how the connection with the broker will be maintained. Within each connection, one or more sessions are established. Each session is used for a unique delivery thread for messages that are delivered to and sent from the client application. This chapter explains the programming required to establish and maintain client connections to brokers through sessions.

A JMS **Session** object represents a single thread of activity. All actual messaging is done through a **Session** object. A **Session** is a factory for **MessageConsumer** and **MessageProducer** objects, each of which remains associated with the **Session** throughout its lifespan. A **Session** is associated with the **Connection** object that creates it.

A **Connection** provides a **createSession()** method for creating a **Session**. This method can be called multiple times to create multiple **Session** objects, each of which remains associated with the **Connection** throughout its lifespan. The signature of the **createSession()** method is as follows:

```
javax.jms.Session createSession(boolean transacted, int acknowledgeMode)
```

where:

- **transacted** — [true | false]

If **true**, the session will be transacted.

- **acknowledgeMode** — [AUTO_ACKNOWLEDGE | CLIENT_ACKNOWLEDGE | SINGLE_MESSAGE_ACKNOWLEDGE | DUPS_OK_ACKNOWLEDGE]

Indicates whether the client will acknowledge any messages it receives.

AUTO_ACKNOWLEDGE, **CLIENT_ACKNOWLEDGE**, and **DUPS_OK_ACKNOWLEDGE** are defined in **javax.jms.Session**. **SINGLE_MESSAGE_ACKNOWLEDGE** is defined in **progress.message.jclient.Session**.

The parameters of a **Session** are qualified so that when the **Session** is transacted, the **acknowledgementMode** setting has no effect, because the transaction implicitly handles acknowledgement. Similarly, **acknowledgementMode** has no effect when a **Session** is only producing messages.

Naming Sessions

A named session can help an administrator identify sessions. Session names do not need to be unique—they are only information labels. The name is set when the session is created, and cannot be changed.

The additional **createSession()** methods enable you to associate a name with a session that will be exposed in session information, as in the **Manage > Broker > Connections** panel in the Sonic Management Console.

The signature of this **createSession()** method is as follows:

```
progress.message.jclient.Connection.createSession (boolean transacted, int  
acknowledgeMode, String sessionName)
```

where:

- **transacted** — [true | false]

If **true**, the session will be transacted.

- **acknowledgeMode** — [AUTO_ACKNOWLEDGE | CLIENT_ACKNOWLEDGE | SINGLE_MESSAGE_ACKNOWLEDGE | DUPS_OK_ACKNOWLEDGE]

- **sessionName** — A String that contains characters that are valid for **ClientID**. See [Connection Factories](#) on page 117.

The value can be null or an empty String—in which case, the behavior is the same as the standard API without **sessionName**.

To retrieve a session's name, use:

```
String progress.message.jclient.Session.getSessionName()
```


The following code excerpt creates a session named **SendUpdates**, and then displays the name of the resulting session:

```
javax.jms.Connection conn;
javax.jms.Session sess;
conn = ...obtain connection from connection factory...
sess = ((progress.message.jclient.Connection)conn).createSession(false,
javax.jms.Session.AUTO_ACKNOWLEDGE, "SendUpdates");
System.out.println("Session name: " +
((progress.message.jclient.Session)sess).getSessionName());
```

Note: Corresponding **TopicSession**, **QueueSession**, and XA methods that enable named sessions are defined in the API.

Acknowledgement Mode

Communication between the broker and the message consumer involves an indication of receipt of the message. One of the following acknowledgement modes is enforced for all messages in a session:

- **AUTO_ACKNOWLEDGE** — The session automatically acknowledges the client's receipt of a message before the next call to receive (**synchronous** mode) or when the session **MessageListener** successfully returns (**asynchronous** mode). In the event of a failure, the last message might be redelivered.
- **CLIENT_ACKNOWLEDGE** — An explicit **acknowledge()** on a message acknowledges the receipt of all messages that have been produced and consumed by the session that gives the acknowledgement. In the event of a failure, all unacknowledged messages might be redelivered.
- **SINGLE_MESSAGE_ACKNOWLEDGE** — An explicit **acknowledge()** on a message acknowledges only the current message and no preceding messages. In the event of a failure, all unacknowledged messages might be redelivered. This mode is a SonicMQ extension to the JMS standard.
- **DUPS_OK_ACKNOWLEDGE** — The session “lazily” acknowledges the delivery of messages to consumers, possibly allowing multiple deliveries of messages after a system outage.

warning: While acknowledgement sets standards for delivery from the client to the broker, there is no reply to the sending application. If an application requires a reply to the sender, use the **JMSReplyTo** header field to indicate the request and program your application to respond to this header field. The requestor can also append a correlation identifier that will ensure that the reply matches its request.

Recover

A client might build up a large number of unacknowledged messages while attempting to process them. A session'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 need for the **recover()** method is most apparent when the acknowledgement mode is **CLIENT_ACKNOWLEDGE** or **SINGLE_MESSAGE_ACKNOWLEDGE**.

Limiting Redelivery from Queues

An application could get into a loop where it repeatedly receives a message that causes the application to fail and rollback the transaction, and then the same message is redelivered. An infinite redelivery loop is sometimes referred to as a “poison message scenario.”

Point-to-point consumer clients that want to constrain redelivery attempts can limit the number of deliveries of a message to the consumer by specifying a parameter on the `ConnectionFactory`. Messages that have exceeded the redelivery limit and have not been acknowledged will be processed according to properties specified in the message or will be discarded. If the message property **JMS_SonicMQ_preserveUndelivered** is set to **true**, the message will be placed on the **SonicMQ.DeadMessage** queue (or an alternate destination specified by the **JMS_SonicMQ_destinationUndelivered** property), and the message property **JMS_SonicMQ_undeliveredReasonCode** will be set to the error code **progress.message.jclient.Constants.UNDELIVERED_DELIVERY_LIMIT_EXCEEDED**. If the property **JMS_SonicMQ_notifyUndelivered** is set to **true**, a notification will be sent. If the 'preserveUndelivered' property is not set, the message will be discarded.

See [Handling Undelivered Messages](#) and [Specifying a Destination for Undelivered Messages](#) for more information.

Note: Alternatively, JMS applications can perform detection on their own by getting and acting on the value of the **JMSXDeliveryCount** property on each message.

The circumstances under which a message can be redelivered to the consumer depend on the session's acknowledgement mode:

- **AUTO_ACKNOWLEDGE** or **DUPS_OK_ACKNOWLEDGE** — Nontransacted sessions that choose **AUTO_ACKNOWLEDGE** or **DUPS_OK_ACKNOWLEDGE** acknowledgement have messages redelivered to a consumer when the application's **onMessage()** method throws an exception. The client runtime catches the exception, and then calls **onMessage()** again. Exceptions are caught and reported to the Connection's **ExceptionListener**. Setting a limit to redelivery attempts limits the redelivery count.
- **SINGLE_MESSAGE_ACKNOWLEDGE** or **CLIENT_ACKNOWLEDGE** — For nontransacted sessions that choose **SINGLE_MESSAGE_ACKNOWLEDGE** or **CLIENT_ACKNOWLEDGE** acknowledgement, messages are redelivered when the application calls **Session.recover()**.
- **TRANSACTION** — Messages are redelivered when an application rolls back the transaction.

The JMS defined property **JMSXDeliveryCount** uses an **int** to specify the number of delivery attempts for a message. The value of this property is incremented every time a message is given to a consumer.

Delivery counters are maintained in the client runtime for messages waiting to be delivered to a consumer object. Applications for which redelivery limit detection is effective are those that create long-lived Consumers: in other words., **ConnectionConsumers**, or **MessageConsumers** that are created once and reused. If a consumer is closed and recreated, the counter for each message sent to the consumer is reset to **0**.

Setting Maximum Delivery Count

By setting the value of the maximum delivery count, you can specify:

- **0**, the default value, which means that there is no redelivery limit
- **1** or more, which means to deliver and then redeliver the specified number of times

For more information about setting and getting the maximum delivery count for a PTP receiver:

- As set programmatically for a Point-to-point receiver on a `ConnectionFactory`, see [This option cannot be set on Connection Factories that are defined as Administered Objects](#).
- As set administratively in a JMS Administered Object, see the “JMS Administered Objects Tool” chapter in the *Aurea SonicMQ Configuration and Management Guide*.

Explicit Acknowledgement

The **Message** interface provides an **acknowledge()** method, which explicitly acknowledges a message. However, the behavior of this method depends on how the **Session** was created.

When the **Session** acknowledgement mode is:

- **AUTO_ACKNOWLEDGE**, the method is ignored.
- **CLIENT_ACKNOWLEDGE**, the method explicitly acknowledges all unacknowledged messages received so far by the session.
- **DUPS_OK_ACKNOWLEDGE**, the method is ignored.
- **SINGLE_MESSAGE_ACKNOWLEDGE**, the method explicitly acknowledges the current message.

When the **Session** is transacted, the method is ignored.

Transacted Sessions

When a **Session** is **transacted**, that **Session** will combine a group of one or more messages with client-to-broker **ACID** properties: **Atomic**, **Consistent**, **Isolated**, and **Durable**.

When a **Session** is transacted, 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 in the ways described in the following table:

Table 10: Transacted Session Events by Message Role (Continued)

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 done 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 are closed without finishing the transaction
- The application, connection, or broker experience 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

In an implementation, an undetected hang occurring in a session thread can lead to unexpected behavior. A message that is 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 or released so that it can be put back on its queue. For message producers, the message is not accessible to a receiver so that it might be further processed.

A SonicMQ broker can use a broker configuration property to indicate that it will not tolerate transacted messages that have been 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 from 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** (an indicator 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 take best efforts to complete transactions as soon as possible.

Without broker-managed timeouts, the transaction will still rollback when the application disconnects or shuts down.

Distributed Transactions

When transactions are contained within a session, the transaction is on a single communication with a broker. The control of the transaction is entirely local.

More sophisticated transactions arise where two sessions enclose the complete transaction. In such cases, applications can implement X/Open's XA protocol to enable transaction identification and transaction demarcation. These global transactions can be further abstracted by interfacing with a transaction manager.

Distributed transactions are discussed in [Distributed Transactions Using XA Resources](#) .

Duplicate Message Detection

The broker can be set up to commit transactions such 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 **javax.jms.TransactionRolledBackException** 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 on a broker from the Sonic Management Console by selecting the **Broker Properties** and, under the **Advanced** tab, clicking **Edit** in the **Properties** section.

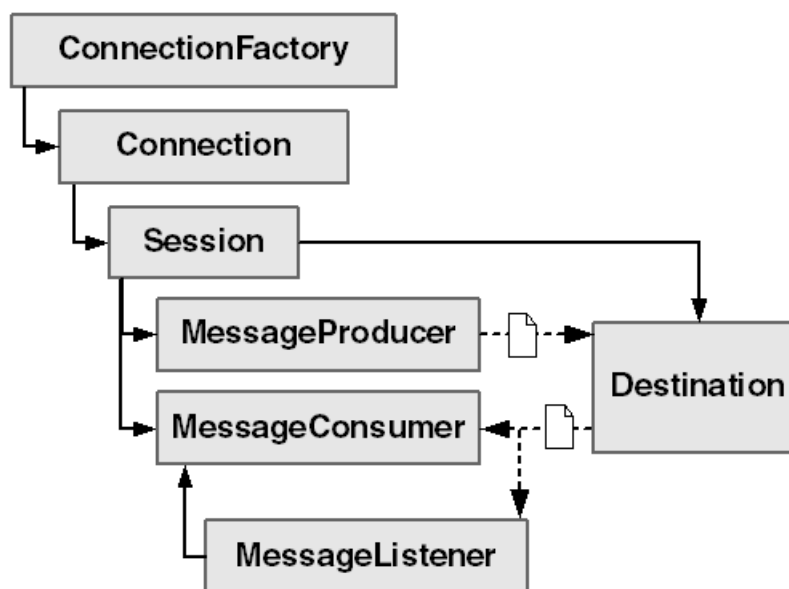
You can alternatively use a hashcode calculated over the message payload instead of a UUID for the **transactionID**. You must ensure that the hashcode is unique for each unique transaction being tracked within the transaction age limit you have set.

See [Duplicate Message Detection Overview](#) for more information about detecting duplicate messages.

Session Objects

The primary session objects allow creation of the destinations, producers, consumers, and messages that are used in the session, as shown in the following figure.

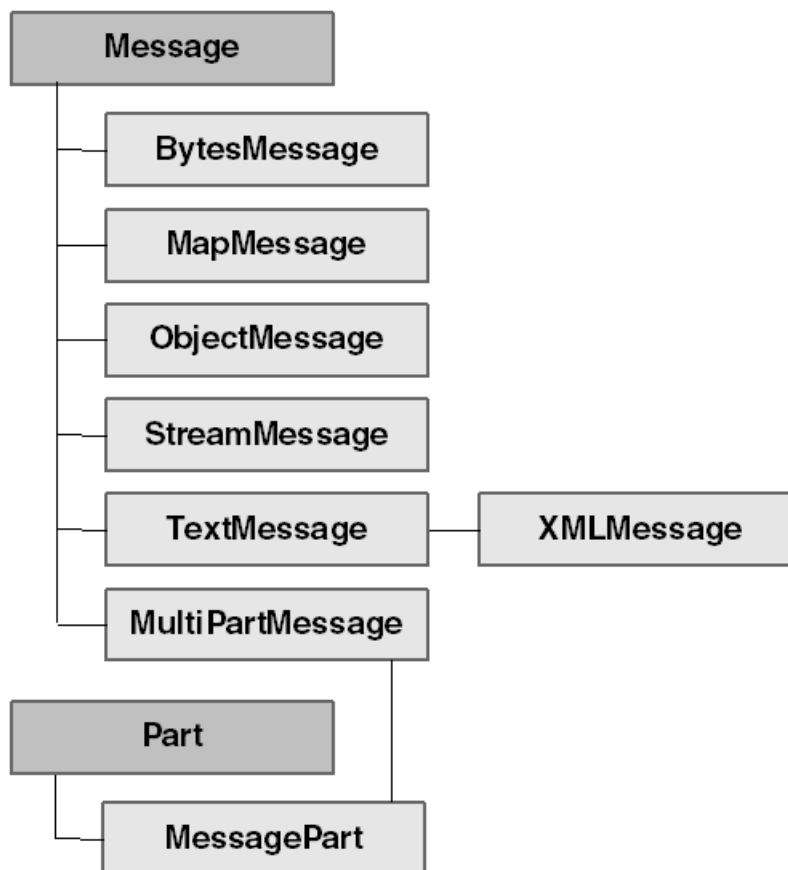
Figure 47: Primary Session Objects



The following figure shows the types of message objects that are created from session methods. The message types are common and extended into both JMS domains.

The **XMLMessage** type is unique to SonicMQ and is an extension of the **TextMessage** type. The **MultipartMessage** type is unique to SonicMQ and is an extension of the **Message** type.

Figure 48: Types of SonicMQ Message Objects



Creating a Destination

Destinations are administered objects that can be controlled by an administrator and can be retrieved through JNDI or other object storage mechanisms.

See “JMS Administered Objects Tool” in the *Aurea SonicMQ Configuration and Management Guide* to learn how the JMS Administered Object tool in the Sonic Management Console allows you to create destinations in both JNDI and file stores.

Important: Security enabled brokers can deny access to destinations. See the chapter “Security Considerations in System Design:” in the *SonicMQ Deployment Guide* for information about access control.

The destination object created can be a queue or a topic.

Destination Objects

There are two destination creation methods:

Point-to-point: createQueue

```
javax.jms.Queue queue = session.createQueue(queueName)
```

where:

queueName is a String name. Its meaning is evaluated from the destination name syntax you use. When the **queueName** is JMS destination, a queue by that name must exist on the broker. If security is enabled on the broker, access control might deny the user from reading or writing to a queue.

Publish and Subscribe: createTopic

```
javax.jms.Topic topic = session.createTopic(topicName)
```

where:

topicName is a String name. Its meaning is evaluated from the destination name syntax you use. If security is defined for topics, the user might be constrained from reading or writing at a topic content node. See [Hierarchical Name Spaces](#) for topic name patterns for subscriptions.

Destination Name Syntax

The syntax of a destination name allows for patterns of JMS destination names and for patterns for routings to remote nodes and URLs:

- Hierarchical structure that enables the use of template characters, as described in [Hierarchical Name Spaces](#).
- Node-qualified names that enable the use of template characters. These names define access to routing definitions and that set permissions to route to specified JMS destinations or URLs.

The following table shows the general syntax for queues (**Q**), topics (**T**), routing nodes (**N**), and HTTP URLs (**U**).

The name you use in the **createQueue** or **createTopic** method is evaluated from its syntax to have one of several meanings.

Table 11: Patterns in Destination Names

Name Description	create Queue	create Topic	Either createQueue or createTopic
Destination name	Q	T	http://U
Destination name with hierarchical structures	Q1.2.3.4	T1.2.3.4	http://a.b.c
Destination name with hierarchical structures and template characters	-	T1.2.*.# (valid for subscribers only)	-
Node-qualified destination name	N::Q	N::T	N::U
Node-qualified destination name with template characters	-	N::# (valid for subscribers only)	-
Node-qualified destination name with hierarchical structures	N::Q.Q.Q	N::T.T.T	N::http://a.b.c
Node-qualified destination name with hierarchical structures and template characters	-	N::T.*.# (valid for subscribers only)	-

If you previously used the **X-HTTP-DestinationURL** technique, you made a construct similar to the following:

```
msg.setStringProperty("X-HTTP-DestinationURL", "http://destinationURL");
sender.send(session.createQueue("sonic.http:foo"), msg);
```

Where **foo** is a placeholder that never gets evaluated. When the default routing `sonic.http` is called for routing, the destination URL was overwritten in the routing definition.

This technique is made obsolete by the ability to supply the URL in the **createQueue** queue name or **createTopic** topic name (which you use for sending to a URL is not important):

```
sender.send(session.createQueue("sonic.http:http://destinationURL"), msg);
```

Effects of Access Control

User names in clients that initiate producer or consumer actions are subject to the broker's authorization policy when the broker has enabled security.

Propagation of ACL Changes in the Broker's Authorization Policy

When administrators adjust Access Control Lists (ACLs), the revised ACLs generally propagate to the broker but do not always propagate to a user's client sessions.

Producer actions reconfirm their access permission at each `send/publish`:

- Clients denied `publish/send` actions discover when they are granted permission.
- Clients granted `publish/send` actions discover when they are denied permission.

Consumer actions have different behavior dependent on the direction of the change:

- Clients denied `subscribe/receive` actions discover when they are granted permission.
- However, clients that were granted `subscribe/receive` actions will not dynamically discover when they are subsequently denied permission in the active session. The client must reconnect (stop and restart) to become aware of this change in permission.

Rechecking ACLs on Messages held for Durable Subscribers

When a message is delivered to a disconnected durable subscription, the ACL is checked at the time the message is held for the subscription. When the durable subscriber reconnects, the authorization is not rechecked at the message restoration. If the subscriber's authorization had changed during the disconnected period, the message would still be delivered.

Because some deployments might not want that behavior, you can set the broker advanced property **BROKER_SECURITY_PARAMETERS.ENABLE_ACL_CHECK_AT_RESTORE** to **true**, to require that access control is rechecked upon restoration.

Temporary Queues

A **TemporaryQueue** object is a unique **Queue** object created for the duration of a **Connection**. It is a system-defined queue that can be consumed only by the **Connection** that created it. A **TemporaryQueue** object can be created at either the **Session** or **QueueSession** level:

```
Session.createTemporaryQueue()
QueueSession.createTemporaryQueue()
```

Creating it at the **Session** level allows to the **TemporaryQueue** to participate in transactions with objects from the Pub/Sub domain. If it is created at the **QueueSession**, it will only be able participate in transactions with objects from the PTP domain.

Unique temporary queue names are generated internally by SonicMQ with values that do not define a use or a queue. A typical temporary queue name is shown below (as one line):

```
$ISYS.USERS.TemporaryQueues.Administrator.$TMPAPPID$7$$CONNECTION$.*.*.11332587362311944962118NodeA
```

Temporary Queues Can Have An Embedded Name Tag

SonicMQ provides metrics on temporary queues. In order to filter relevant temporary queues, an overload of the **Session.createTemporaryQueue()** method lets you supply a **customID** String.

```
Session.createTemporaryQueue(String customID);
```

This method embeds the user-supplied **customID** at some position in the temporary queue name—there is no guarantee exactly where in the name. For example the following code:

```
mySession.createTemporaryQueue("CreditCheckReplyQueue");
```

creates a temporary queue with a name similar to the following (as one line):

```
$ISYS.USERS.TemporaryQueues.Administrator.$TMPAPPID$3$$CONNECTION$.*.*.491974464CreditCheckReplyQueue2307944962118NodeA
```

Notice that the customID is embedded in the middle of the temporary queue name.

Important: Limited Length of Temporary Queue Name — Temporary queue names are restricted to 256 characters. As temporary queue names without a **CustomID** are rather long, keep your assigned custom identifier brief.

See the “Instance Metrics” section of the “Monitoring the Sonic Management Environment” chapter of the *Aurea SonicMQ Configuration and Management Guide* to see how metrics are set on temporary queues.

Temporary destinations (**TemporaryTopic** or **TemporaryQueue**) can be created for request-and-reply mechanisms. See [Reply-to Mechanisms](#) for more information.

Using a Lookup for Destinations

While topics and queues are administered objects, there are advantages to programmatic lookup of defined destinations.

SonicMQ lets you store administered objects in some object store—JNDI or a simple file store—and then reference the object indirectly (by name) in some context.

See *Lookup Using the Sonic JNDI SPI* 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. The signature for this method is:

```
public java.jms.MessageProducer createProducer(java.jms.Destination destination)
    throws JMSException
```

Queue and **Topic** both inherit from **Destination**, so 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 javax.jms.MessageConsumer createConsumer(javax.jms.Destination destination) throws JMSEException**
- **public MessageConsumer createConsumer(javax.jms.Destination destination, java.lang.String messageSelector) throws JMSEException**
- **public MessageConsumer createConsumer(javax.jms.Destination destination, java.lang.String messageSelector, boolean NoLocal) throws JMSEException**

Since both **Queue** and **Topic** inherit from **Destination**, either is a valid **Destination**.

The **MessageConsumer** object returned by these methods is dedicated to the **Destination** you provide. If the **MessageConsumer** is created with a **Queue**, it honor the JMS semantics for the P2P messaging model; if a **Topic**, the Pub/Sub messaging model.

If you want to create a **MessageConsumer** that is durable subscriber to a **Topic**, you call one of the **Session** object's **createDurableSubscriber()** methods:

- **public javax.jms.TopicSubscriber createDurableSubscriber(javax.jms.Topic, java.lang.String name) throws JMSEException**
- **public javax.jms.TopicSubscriber createDurableSubscriber(javax.jms.Topic, java.lang.String name) java.lang.String messageSelector, boolean NoLocal) throws JMSEException**

Since **TopicSubscriber** inherits from **MessageConsumer**, you can assign the returned **TopicSubscriber** to a **MessageConsumer** reference; this allows you to use the **MessageConsumer** interface to manipulate the object, rather than using the **TopicSubscriber** interface, which might be deprecated in future JMS versions.

Creating a Message

The message type is created from a **Session** method in the general form:

```
javax.jms.[type]Message msg = sendSession.create[type]Message( )
```

where **type** is the JMS message type:

- **javax.jms.TextMessage msg = sendSession.createTextMessage()**
- **javax.jms.BytesMessage msg = sendSession.createBytesMessage()**
- **javax.jms.MapMessage msg = sendSession.createMapMessage()**
- **javax.jms.Message msg = sendSession.createMessage()**
- **javax.jms.ObjectMessage msg = sendSession.createObjectMessage()**
- **javax.jms.StreamMessage msg = sendSession.createStreamMessage()**

The **XMLMessage** and **MultipartMessage** types are SonicMQ extensions to the JMS standard. You cannot create them from a **javax.jms.Session**, because the required methods are not defined for that interface. However, you can cast the **javax.jms.Session** to a **progress.message.jclient.Session** first, as shown:

```
progress.message.jclient.Session pSendSession;  
progress.message.jclient.XMLMessage xMsg;  
progress.message.jclient.MultipartMessage multipartMsg;  
pSendSession = (progress.message.jclient.Session) sendSession;  
xMsg = xSendSession.createXMLMessage( );  
mutipartMsg = xSendSession.createMultipartMessage( );
```

See [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 some other session method is being executed in another thread.

Closing a **CLIENT_ACKNOWLEDGE** session does not force an **acknowledge()** to occur. Attempts to use a closed connection's session objects throws 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.

Note: Close Timeout under Asynchronous Message Delivery — Asynchronous message delivery can be set in the connection factory to provide performance improvements, particularly for replicating brokers. When asynchronous delivery mode is enabled, some messages in client buffers might not have been delivered to (or acknowledged by) the broker. When messages are pending delivery and **close** is called, producers that are flow controlled or clients with a backlog of messages, **close** could take a while. Applications unwilling to wait can configure a **close** timeout. See [Asynchronous Message Delivery](#) and [Close Behavior](#) for more information.

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 will throttle 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 that when resources are nearly exhausted, SonicMQ can, under programmatic control, throw exceptions until flow control conditions are cleared.

When flow control is active, the messages might be sent to consumers at a rate that is faster than that at which 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.
- Set the DeliveryMode on messages to **DISCARDABLE**.

Note: Messages sent to a queue will only expire after they have been 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.

Using Client Persistence and Wait Time When Flow Controlled

Clients using persistent client functionality can configure the persistent client to write messages to the local store when a message producer is flow-controlled. Then, when producer flow control is no longer in effect, persisted messages flow to the broker in order while the message producer continues to add messages to the local store. When the local store is cleared, messages flow directly from the producer to the broker. The application sender can set the wait time before messages paused by flow control are written to the local store.

The persistent client controls the rate of accepting messages into the store relative to the rate of sending stored messages out of the local store to the broker in an effort to drain the backlog of messages. The sender experiences a slower producer rate while messages are being restored. However, it is possible for messages to accumulate in the store faster than they can be sent to the broker. If this occurs, the local store size might be exceeded in which case the sender gets an exception.

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—and corrective action 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(java.lang.Integer interval)**. Where interval takes a value as follows:

- `progress.message.jclient.Constants.MONITOR_INTERVAL_USE_BROKER_SETTING (-1)` - the Client Default Monitor Interval defined on broker or cluster is used. Supported only for clients and brokers using SonicMQ 2013 or later, where this is the default.
- **0** - Flow control notifications are disabled.
- **>=1** - Flow control monitoring interval in seconds.

The monitoring interval is determined when a connection is established and applies to all sessions created by that connection. It cannot be modified during the lifetime of the connection.

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 has been blocked for one full monitoring interval, it might take as long as another monitoring interval from the time the session became 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**, **JMS ClientID**, and JMS 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.

Notification Interface

Notifications are collected and displayed in the Sonic Management Console and delivered to any management client that has registered an appropriate notification listener (see the *Aurea SonicMQ Administrative Programming Guide* for more information). This interface has a callback that handles all notification types.

To view flow control notifications in the Sonic Management Console, select the **Containers** node in the **Manage** view. Under the container instance node, right-click the broker instance where you want to view the notifications. In the window that opens, select the flow control events under the **Applications** node. See the *Aurea SonicMQ Configuration and Management Guide* for more information about viewing flow control notifications in the Sonic Management Console.

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 will not be active in the session.

Flow to Disk

If flow control is active, message producers may block, while waiting for message consumers to process messages that have 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 *Aurea 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`). As an application programmer, you can explicitly override the broker setting.

To override this setting for all sessions, call the following method:

```
ConnectionFactory.setFlowToDisk(Integer flowSetting)
```

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

- `progress.message.jclient.Constants.FLOW_TO_DISK_USE_BROKER_SETTING` (the default) — Specifies that the broker setting of `FLOW_TO_DISK` will be used for the consumer.
- `progress.message.jclient.Constants.FLOW_TO_DISK_ON` — Specifies that `FLOW_TO_DISK` is turned on for the consumer regardless of the broker setting.
- `progress.message.jclient.Constants.FLOW_TO_DISK_OFF` — Specifies that `FLOW_TO_DISK` is turned off for the consumer regardless of the broker setting.

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

```
Session.setFlowToDisk(int 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 `Session.setFlowToDisk()` method will have no effect.

Send Timeout

A `MessageProducer`'s `send()` call may block, for example due to flow control or while handling a fault tolerant reconnect. Setting a send timeout allows the client application to regain control if `send()` blocks for too long. It provides an alternative to disabling flow control entirely.

The send timeout is configured via the `ConnectionFactory` or on an individual `Session` and applies to all `MessageProducers` associated with the `Session`:

- `public void progress.message.jclient.ConnectionFactory.setSendTimeout(int milliseconds)`
- `public void progress.message.jclient.Session.setSendTimeout(int milliseconds)`

If the send timeout is specified via the `ConnectionFactory` it becomes the default for all `Sessions`, but can still be overridden at the `Session` level. The send timeout can be changed over the `Session`'s lifetime if needed.

If the send timeout fires, `MessageProducer.send()` throws a `progress.message.jclient.JMSSendTimeoutException`. The message that was being sent is then in doubt – it may or may not be delivered.

Applications should *avoid* tearing down and recreating their `Connections` when handling a `JMSSendTimeoutException`. Doing so would be unnecessarily expensive given that:

1. the condition blocking the send is expected to clear, and
2. reconnecting would typically put you back in the same state with the next `send()` blocking, which in turn could trigger another reconnect, and so on.

If the connection itself fails then the `Connection's` `ExceptionListener` is called, at which point the application can attempt to reconnect.

Transacted sessions do not support send timeout. For non-transacted sessions, the send timeout behaves as follows:

Send Timeout is not available for Transacted Sessions or Asynchronous Message Delivery.

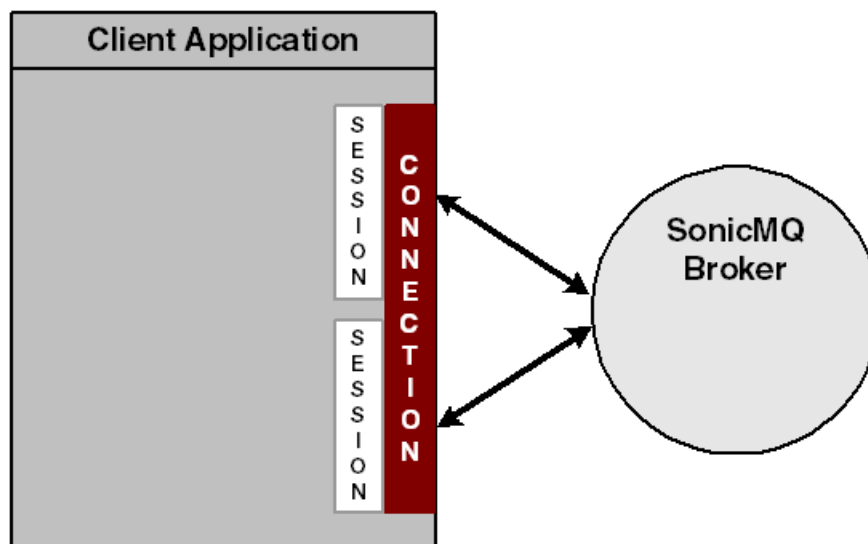
Using Sessions and Consumers

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

Using multiple sessions gives up the benefits of serialized operations on a single thread of execution. Multiple sessions are best suited for alternate or supporting functions within an application. The following figure 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 49: Multiple Sessions on a Connection



Creating Session Objects and the Listeners

The sections [Creating and Monitoring a Connection](#) and [Handling Exceptions on the Connection](#) provide information and examples of setting up connections. Once you have a connection, you can create session objects and listeners.

In the following code snippet, a continuation of the **ReliableChat** sample shown in the section [Creating and Monitoring a Connection](#), 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.

ReliableChat: Create Session Objects and Listeners

```

pubSession = connect.createSession(false, javax.jms.Session.CLIENT_ACKNOWLEDGE);

subSession = connect.createSession(false, javax.jms.Session.CLIENT_ACKNOWLEDGE);

javax.jms.Topic topic = pubSession.createTopic (APP_TOPIC);
javax.jms.MessageConsumer subscriber = subSession.createDurableSubscriber(topic,
    "SampleSubscription");
subscriber.setMessageListener(this);
publisher = pubSession.createProducer(topic);
// Register this class as the exception listener for any problems.
connect.setExceptionListener( (javax.jms.ExceptionListener) this);
  
```


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](#) for information about setting up and working with connections.

JMS Messaging Domains

The JMS messaging domains are primarily differentiated by messaging behaviors. The programming functionality for the domains is similar, as shown in the interfaces and methods in the following table.

Table 12: Connected Session Functionality Common to PTP and Pub/Sub

javax.jms Interface	Functionality in Either Domain
ConnectionFactory	<ul style="list-style-type: none"> Allows administrative control of communication resources Creates one or more Connections
Connection	<ul style="list-style-type: none"> 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
Session	<ul style="list-style-type: none"> 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

Integration with Application Servers

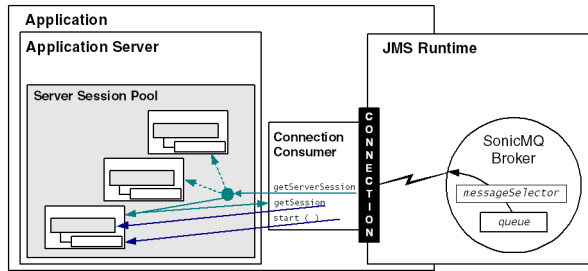
Application servers are capable of handling multiple sessions concurrently, offering high availability of the application. By creating and maintaining a server session pool, the session thread wrapped in each server session can be started, used, then stopped and returned to the pool when it has completed its work.

Connection Consumer

An application server creates a `ConnectionConsumer` to asynchronously receive messages and pass them to a `ServerSessionPool` where the messages are assigned to server sessions.

The `ConnectionConsumer` receives messages through the connection for the destination it specified, filtering the preferred messages through its message selector, then distributes the message to sessions, as shown in the following figure. This behavior enables the consumer's messages to be processed concurrently by several sessions.

Figure 50: ServerSession Pool and Connection Consumer



The `create` method for the `ConnectionConsumer` indicates the `ServerSessionPool` that is an object implemented by an application server to provide a pool of `ServerSessions` for processing the messages of the `ConnectionConsumer`.

You create a `ConnectionConsumer` object by calling the `Connection` object's `createConnectionConsumer()` or `createDurableConnectionConsumer()` methods:

- ```

public javax.jms.ConnectionConsumer
createConnectionConsumer(javax.jms.Destination destination, java.lang.String
messageSelector, javax.jms.ServerSessionPool sessionPool,int maxMessages)
throws JMSEException

```
- ```

public javax.jms.ConnectionConsumer
createDurableConnectionConsumer(javax.jms.Topic topic, java.lang.String
subscriptionName, java.lang.String messageSelector, javax.jms.ServerSessionPool
sessionPool, int maxMessages) throws JMSEException

```

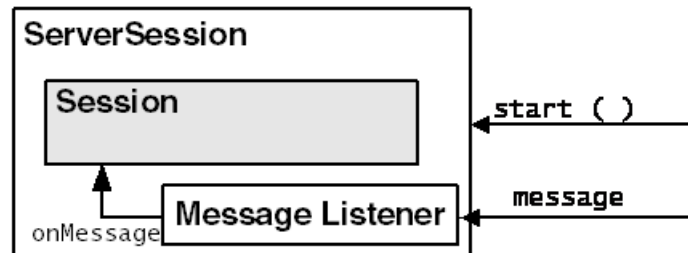
where:

- `destination` is the `Queue` or `Topic` to access
- `topic` is the `Topic` to access
- `messageSelector` is the `String` with the message selector definition
- `sessionPool` is the `ServerSessionPool` to associate with this connection consumer
- `subscriptionName` is the name of the durable subscription
- `maxMessages` is the maximum number of messages that can be assigned to a server session at one time

Server Session

A connection consumer executes a **getServerSession()** method to return a **ServerSession** from the pool. A **ServerSession** is an application server object that associates a thread with a JMS session. It offers two methods, **getSession()** to return the **ServerSession's** JMS session, and **start()** to start the execution of the **ServerSession** thread that results in the execution of the JMS Session's **run()** method.

Figure 51: Server Session for a Connection Consumer to a Queue



A **ServerSession** wraps a **Session** and associates a **MessageListener**. As shown in the following figure, the **ServerSession** is sent a message obtained by the **ConnectionConsumer**. The **Session** wrapped by the **ServerSession** is then started so that it can perform its **onMessage()** work.

The **ServerSession** will register some object it provides as the **ServerSession's** thread run object. The **ServerSession's** start method will call the thread's **start()** method, which will start the new thread, and from it, call the run method of the **ServerSession's** run object. This object will do some housekeeping and then call the **Session's** run method. When **run()** returns, the **ServerSession's** run object can return the **ServerSession** to the **ServerSessionPool**, and the cycle starts again.

Message Driven Beans

Connection consumers and server session pools are expert facilities that provide a way to send nonblocking and asynchronous messages to application servers. This functionality is facilitated in Enterprise JavaBeans (EJB) through Message Driven Beans (MDB) of J2EE derived from the interface **javax.ejb.MessageDrivenBean**, which is in turn derived from the **javax.jms.MessageListener**. The **onMessage()** method inherited from the **javax.jms.MessageListener** interface has the sole parameter **javax.jms.Message**.

After the container accesses a bean from a pool of available instances, the received message is passed to the **onMessage()** method of the MDB instance. When the **onMessage()** method completes, the bean is returned to the pool of available instances.

Shared Subscriptions

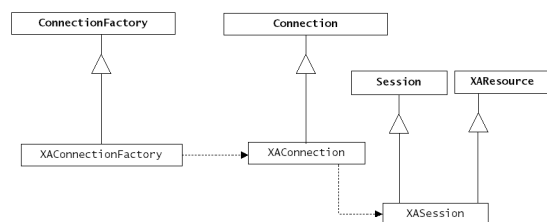
SonicMQ allows shared subscriptions for topics across multiple application servers. Server session pools can be used in combination with shared subscriptions to allow round-robin delivery between application servers, which, in turn, allows round-robin between members of the server session pool.

XA Resources

Distributed transactions, discussed in [Distributed Transactions Using XA Resources](#) require **XAResources** so that they can integrate with a Transaction Manager and application servers.

The following figure describes SonicMQ XA interface objects.

Figure 52: SonicMQ Implementation of the XA Interfaces



The connections and sessions for XA interfaces are similar relationships to those in the standard interface. Some examples of object relationships are:

- The **XASession** is created by the **XAConnection**.
- The **XASession** inherits from the **Session**.

Messages

This chapter provides information about creating and handling messages in SonicMQ. For details, see the following topics:

- [About Messages](#)
- [Message Type](#)
- [Working With Messages That Have Multiple Parts](#)
- [Message Structure](#)
- [Message Header Fields](#)
- [Message Properties](#)
- [Message Body](#)

About Messages

A SonicMQ® message is a package of bytes that encapsulates the message body as a payload and then exposes metadata that identify, at a minimum, the message and its timestamp, destination, and priority. The **instanceof** the object identifies the type of JMS message.

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

```
private void jmsPublish (String aMessage)
{
    javax.jms.TextMessage msg = session.createTextMessage();
    msg.setText( user + ": " + aMessage );
    publisher.publish( msg );
}
```

When a message is received it might be through an asynchronous listener, coded as shown:

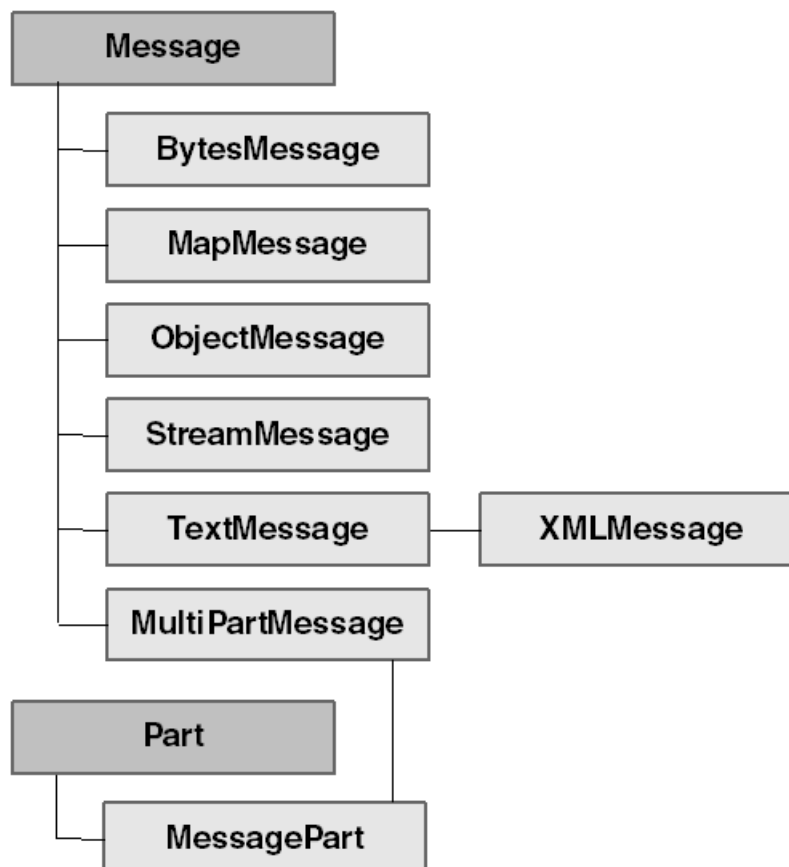
```
// Handle an asynchronously received message
public void onMessage( javax.jms.Message aMessage)
{
    ...
    // Cast the message as a text message.
    javax.jms.TextMessage textMessage = (javax.jms.TextMessage) aMessage;

    // Read a single String from the text message, print to stdout.
    String string = textMessage.getText();
    ...
}
```

Message Type

The JMS specification defines five types of messages, all derived from the **Message** interface, which also defines message headers and the **acknowledge()** method used by all JMS messages. SonicMQ provides an **XMLMessage** type as an extension of the JMS **TextMessage**. The following figure lists the SonicMQ message types.

Figure 53: SonicMQ Message Types



The message types are defined as follows:

- **Message** — The root interface of all JMS 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 will be shared with applications that cannot read Java 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 Java 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 Java object. An **ObjectMessage** is useful when both JMS clients are Java applications or applets with access to the same class definition.
- **StreamMessage** — The body is a stream of Java 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 **java.lang.String** or **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 is a **TextMessage** with XML tags that can be parsed as a valid XML DOM tree or evaluated through SAX.
- **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**, **javax.jms.Message** implementations in addition to primitive types such as XML, HTML, or any type in MIME format such as **text/xml**.

Note: Large message support through recoverable file channels can use any type of message as the header message. The file transfer is performed through the untyped file channel. See [Recoverable File Channels](#)

Creating a Message

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

```
javax.jms.[type]Message msg = session.create[type]Message()
```

Use the following session methods to create the different message types:

- **javax.jms.Message msg = session.createMessage()**
- **javax.jms.BytesMessage msg = session.createBytesMessage()**
- **javax.jms.MapMessage msg = session.createMapMessage()**
- **javax.jms.ObjectMessage msg = session.createObjectMessage()**
- **javax.jms.StreamMessage msg = session.createStreamMessage()**
- **javax.jms.TextMessage msg = session.createTextMessage()**

The **MultipartMessage** type, described on page [Composition of a MultipartMessage](#) on page 197, extends the **Message** type. The **XMLMessage** type, described in the following section, extends the **TextMessage** type.

Working with XML Messages

TextMessage inherits methods from **Message** and adds a text message body. **XMLMessage** then parses XML text through an implementation of JAXP classes.

The Apache Xerces parser is loaded by default but any XML conformant parser can be set as the preferred parser. The Apache Xerces parser supports the XML 1.0 recommendation and contains advanced parser functionality such as XML Schema, DOM Level 2 version 1.0 and SAX Version 2 in addition to supporting DOM Level 1 and SAX version 1 APIs.

JAXP Support

The Java API for XML Parsing (JAXP) is the JavaSoft standard for a Java application to access an XML-conformant parser. An application can swap XML parsers to move between high performance and memory conservation without changing application code.

To use a different compliant SAX or DOM parser, pass the system property in a command line as shown in these command line examples:

- `java -Djavax.xml.parsers.SAXParserFactory= org.apache.crimson.jaxp.SAXParserFactoryImpl myApp`
- `java -Djavax.xml.parsers.DocumentBuilderFactory= org.apache.crimson.jaxp.DocumentBuilderFactoryImpl myApp`

JAXP Interfaces

JAXP provides the following interfaces:

- **DocumentBuilder** — The Document Builder defines the API to obtain DOM Document instances from an XML document. Using this class, you can get a **org.w3c.dom.Document** from XML tagged text. An instance of this class is obtained from the **DocumentBuilderFactory.newDocumentBuilder** method. Then XML can be DOM parsed from a variety of input sources including InputStreams, files, URLs, and SAX InputSources.
- **DocumentBuilderFactory** — The Document Builder Factory defines a factory API that lets applications get a parser that produces DOM object trees from XML documents. The system property that controls the Factory implementation to create is **javax.xml.parsers.DocumentBuilderFactory**. The property names a class that is a concrete subclass of this abstract class. If none is defined, the default is used. When an application has a reference to a **DocumentBuilderFactory** it can use the factory to configure and obtain parser instances.
- **SAXParser** — The SAXParser defines the API that wraps an **org.xml.sax.Parser** implementation class. Using this class allows an application to parse content using the SAX API. An instance of this class can be obtained from the **SAXParserFactory.newSAXParser** method. When an instance of this class is obtained, XML can be parsed from a variety of input sources. Then XML can be SAX parsed from input sources including InputStreams, files, URLs, and SAX InputSources.
- **SAXParserFactory** — The SAX Parser Factory defines a factory API that lets applications configure and obtain a SAX-based parser to parse XML documents. The system property that controls which Factory implementation to create is **javax.xml.parsers.SAXParserFactory**. The property names a class that is a concrete subclass of this abstract class. If none is defined,

the default is used. When an application has a reference to a **SAXParserFactory**, it can use the factory to configure and obtain parser instances.

The following table describes the methods you can use to parse XML messages.

Table 13: Methods for XML Parsing (Continued)

Method	Description
<code>void setDocument(org.w3c.dom.Document aDoc)</code>	Takes the <code>org.w3c.dom.Document</code> <code>aDoc</code> and stores it as the internal document for this message.
<code>void setNamespaceAware(boolean aware)</code>	Set whether or not the underlying <code>javax.xml.parsers.DocumentBuilderFactory</code> used to generate <code>org.w3c.dom.Document</code> returned by <code>XMLMessage.getDocument()</code> is namespace aware.
<code>boolean isNamespaceAware()</code>	Tests whether the underlying <code>javax.xml.parsers.DocumentBuilderFactory</code> used when <code>getDocument()</code> is called is namespace aware.
<code>org.w3c.dom.Document getDocument()</code>	Returns an <code>org.w3c.dom.Document</code> object created from the <code>XMLMessage</code> content that can be accessed by DOM-tree functionality.
<code>void setDocumentBuilderFactory(java.lang.String classname)</code>	Sets the class name for the implementation of JAXP1.1 <code>DocumentBuilderFactory</code> interface to override the default Apache Xerces parser.
<code>void setSAXParserFactory(java.lang.String classname)</code>	Sets the class name for the implementation of JAXP1.1 <code>SAXParserFactory</code> interface to override the default Apache Xerces parser.
<code>java.lang.String getDocumentBuilderFactory()</code>	Gets the class name for the implementation of JAXP1.1 <code>DocumentBuilderFactory</code> interface.
<code>java.lang.String getSAXParserFactory()</code>	Gets the class name for the implementation of JAXP1.1 <code>SAXParserFactory</code> interface.
<code>org.xml.sax.InputSource getSAXInputSource()</code>	Returns an <code>org.xml.sax.InputSource</code> object created from the <code>XMLMessage</code> contents.

DOM Support

The Document Object Model (DOM) provides a tree of objects with interfaces for traversing the tree and writing an XML version of it. The following code snippet, excerpted from the **XMLDOMChat** sample application, provides an example of DOM support.

XMLDOMChat: DOM Support (Continued)

```
public void onMessage(javax.jms.Message aMessage)
{
    try
    {
        // Test the message type.
        if (aMessage instanceof progress.message.jclient.XMLMessage)
        {
            //Cast the message as a XML message.
            progress.message.jclient.XMLMessage xmlMessage =
                (progress.message.jclient.XMLMessage) aMessage;
```

```
//Get the XML document associated with this message.
org.w3c.dom.Document doc = xmlMessage.getDocument();

// Get the sender and content from the message.
org.w3c.dom.NodeList nodes = null;
nodes = doc.getElementsByTagName("sender");
String sender = (nodes.getLength() > 0) ?
nodes.item(0).getFirstChild().getNodeValue() : "unknown";
nodes = doc.getElementsByTagName("content");
String content = (nodes.getLength() > 0) ?
nodes.item(0).getFirstChild().getNodeValue() : null;

//Show the message
System.out.println("[XML from'" + sender + "'" + content);

// Show the message as a tree
printDocNodes(doc.getDocumentElement(),0);
System.out.println();
}
else
{
//Cast the message as a text message and display it.
javax.jms.TextMessage textMessage = (javax.jms.TextMessage) aMessage;
System.out.println("[TextMessage]" + textMessage.getText());
}
}
catch...
```

SAX Support

SAX (Simple API for XML) provides an event-driven mechanism for parsing XML which is optimized for parsing large XML documents. This is the protocol that most servlets and network-oriented programs use to transmit and receive XML documents because of its speed in a modest memory footprint.

However, the SAX protocol requires more program logic than the Document Object Model (DOM). As an event-driven model, SAX is more obscure—you provide the callback methods and the parser invokes them as it reads the XML data. Also, you cannot reposition in or rearrange the document as it is interpreted in a serial data stream.

If your application calls for modifying and displaying an XML document, the DOM is better suited to that task.

The **XMLSAXChat** sample excerpt in the following code snippet shows how a publisher sends an XML message and the subscriber calls **getSAXInputSource()** on the message. That method returns an **org.xml.sax.InputSource** (rather than the **org.w3c.dom.Document** returned in the **XMLDOMChat** sample). Event parsing is done on the XML message and the message is printed out to the screen.

XMLSAXChat: SAX Support (Continued)

```
public void onMessage(javax.jms.Message aMessage)
{
    try
    {
// Test the message type.
        if (aMessage instanceof progress.message.jclient.XMLMessage)
        {
//Cast the message as a XML message.
            progress.message.jclient.XMLMessage xmlMessage =
(progress.message.jclient.XMLMessage) aMessage;
//Get the XML SAXInputSource associated with this message.
            org.xml.sax.InputSource is = xmlMessage.getSAXInputSource();
            if (System.getProperty("javax.xml.parsers.SAXParserFactory")== null) {
//make the default be xerces by setting the System property to point to xerces.
                java.util.Properties props = System.getProperties();
```

```

        props.put("javax.xml.parsers.SAXParserFactory",
"org.apache.xerces.jaxp.SAXParserFactoryImpl");
        System.setProperties(props);
    }
    javax.xml.parsers.SAXParserFactory plugfactory =
javax.xml.parsers.SAXParserFactory.newInstance();
    //Load the parser specified in system property.
    javax.xml.parsers.SAXParser saxParser = plugfactory.newSAXParser();
    org.xml.sax.Parser sp = saxParser.getParser();
    sp.setDocumentHandler(this);
    sp.setErrorHandler(this);
    sp.parse(is);
}
else
{
    //Cast the message as a text message and display it.
    javax.jms.TextMessage textMessage = (javax.jms.TextMessage) aMessage;
    System.out.println("[TextMessage]" + textMessage.getText());
}
}
catch ...

```

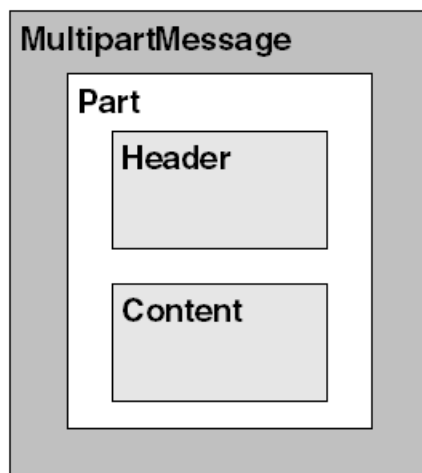
Working With Messages That Have Multiple Parts

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. The JMS specification does not consider messages that contain other messages. SonicMQ uses the DataHandlers in the Java Activation Framework to provide an interface that allows an application to get and set the parts of the message as different types. SonicMQ lets you treat existing JMS messages as parts of a multipart message and even to include one multipart message inside another.

Composition of a MultipartMessage

The structure of a **MultipartMessage** is a wrapper surrounding a series of **Parts**, as shown in the following figure.

Figure 54: MultipartMessage Wraps Parts That Have Header and Content



Each **MultipartMessage** can have JMS 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:

- `progress.message.jclient.MultipartMessage`
- `progress.message.jclient.Part`
- `progress.message.jclient.Header`

MultipartMessage Type

The **MultipartMessage** type is a subclass JMS **Message** and follows the JMS semantics for interactions with sessions, producers and consumers. A **MultipartMessage** is limited to 10 megabytes, 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 from five distinct parts:

- **part1** is a **TextMessage**:

```
javax.jms.TextMessage msg1 = session.createTextMessage();
msg1.setText(" this is a JMS TextMessage " );
Part part1 = mm.createMessagePart(msg1);
part1.getHeader().setContentId("CONTENTID1");
```

- **part2** is a **byte[]**:

```
String str2 = "This string is sending as a byte array";
DataHandler dh = new DataHandler (str2.getBytes(), "myBytes");
Part part2 = mm.createPart(dh);
part2.getHeader().setContentId("CONTENTID2");
```

- **part3** contains simple text:

```
String str1 = "a simple text string to put in part 3";
dh = new DataHandler (str1, "text/plain");
Part part3 = mm.createPart(dh);
part3.getHeader().setContentId("CONTENTID3");
```

- **part4** accesses a file:

```
FileDataSource fds = new FileDataSource("Readme.txt");
dh = new DataHandler (fds);
Part part4 = mm.createPart(dh);
part4.getHeader().setContentId("CONTENTID4");
System.out.println("sending part4..a Readme file");
```

- **part5** is a Web site address:

```
java.net.URL url = new java.net.URL("http://www.cnn.com");
dh = new DataHandler (url);
Part part5 = mm.createPart(dh);
```

- The parts are added to the **MultipartMessage** and then sent:

```
mm.addPart(part1);
mm.addPart(part2);
mm.addPart(part3);
mm.addPart(part4);
mm.addPart(part5);
sender.send(mm);
```

The producer methods in the **MultipartMessage** interface are listed in the following table.

Table 14: Producer Methods in the MultipartMessage Interface (Continued)

Method	Description
<code>Part createPart()</code>	Creates an empty part.
<code>Part createPart(javax.activation.DataHandler handler)</code>	Creates a Part from a DataHandler where handler is the data
<code>Part createPart(java.lang.Object object, java.lang.String type)</code>	Creates a part where type is the ContentType associated with the content.
<code>Part createMessagePart(Message message)</code>	Creates a part whose content is a Message where message is the content of the part.
<code>void addPart(Part part)</code>	Adds part to the multipart at the end of the message.
<code>void addPartAt(Part part, int index)</code>	Adds part to the multipart at position index.
<code>void removePart(java.lang.String cid)</code>	Removes part with content-ID cid from the MultipartMessage.
<code>void removePart(int index)</code>	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 **javax.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 JMS message. Consuming the message is done with standard **MessageListeners** or calls to **receive()**.

In the **MultipartMessage** sample where **onMessage** delivers an **instanceofMultipartMessage**, it 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, as shown:

```
private void unpackMM(javax.jms.Message aMessage, int depth) {
    int n = depth;
    ...
    try
    {
```

```

indent(n);
System.out.println ("Extend_type property = " +
aMessage.getStringProperty(Constants.EXTENDED_TYPE));
MultipartMessage mm = (MultipartMessage)aMessage;
int partCount = mm.getPartCount();
indent(n);

```

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

```

if (mm.isMessagePart(i))
{
    javax.jms.Message msg = mm.getMessageFromPart(i);
    if (msg instanceof MultipartMessage)
        unpackMM(msg, ++depth);
    else
        unpackJMSMessage(msg, n);
}
else
{
    unpackPart(part, n);
}
}

```

The methods for the **MultipartMessage** consumer are listed in the following table.

Table 15: Consumer Methods in MultipartMessage Interface (Continued)

Method	Description
java.lang.String getProfileName()	Return the extended type or profile that was used to create this message
boolean doesPartExist(java.lang.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(java.lang.String cid)	Tests whether if 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(java.lang.String cid)	Gets the part of the message identified as content ID cid
Message getMessageFromPart(int index)	Gets a JMS message from part index of the message
Message getMessageFromPart(java.lang.String cid)	Gets a JMS message from the part of the message with the content ID cid
boolean isReadOnly()	Tests whether a message is read only
void clearReadOnly()	Makes the message writable

JMS_SonicMQ_ExtendedType Property

A MultipartMessage is not only identified as an instance of MultipartMessage. For the convenience of older SonicMQ versions or other JMS providers, 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 **progress.message.jclient.Constants** as:

```
public String EXTENDED_TYPE = "JMS_SonicMQ_ExtendedType"
```


Parts of a MultipartMessage

A part of a **MultipartMessage** follows the design pattern for Java handling of MIME used in JAXM and JavaMail™ through the JavaBeans Activation Framework. Each part has an associated **Header** and content, as described:

- 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 through a **javax.activation.DataHandler** in the following formats:
 - DataHandler by using the **getDataHandler()** method or through a **javax.activation.DataHandler** object. The **DataHandler** object lets you discover the operations available on the content and then instantiate the appropriate component to perform those operations. A **DataContentHandler** class for the specified type must be available to ensure the expected result. For example, **setContent(mycontents, "application/x-mytype")** expects a **DataContentHandler** for **application/x-mytype**.
 - Input stream by using the **getInputStream()** method.
 - Java object by using the **getContent()** method. This method returns the content as a Java object.

The methods in the **Parts** interface are listed in the following table.

Table 16: Methods in the Parts Interface (Continued)

Method	Description
setContent(java.lang.Object object, java.lang.String type)	Sets the part's content as a Java Object of content type type
setContent(byte[] content)	Sets the part's content as a byte array of content
setDataHandler(javax.activation.DataHandler dataHandler)	Specifies the DataHandler to set the part's content by wrapping the actual content
java.lang.Object getContent()	Gets the content of the part as an Object
byte[] getContentBytes()	Gets the content of the part as a byte array
javax.activation.DataHandler getDataHandler()	Provides the mechanism to get this part's content. Returns the DataHandler for the Part
Header getHeader()	Gets the Header for the Part
java.io.InputStream getInputStream()	Invokes the DataHandler's getInputStream() method and returns an input stream for this part's content
java.io.OutputStream getOutputStream()	Invokes the DataHandler's getOutputStream() method and 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 JMS messages into a **MultipartMessage**. The **ContentType** of SonicMQ **MessageParts** is set implicitly, as shown in the following table:

Table 17: Implicit Content-Type for JMS Message Types (Continued)

JMS/SonicMQ 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 (SonicMQ)	application/x-sonicmq-xmlmessage
MultipartMessage (SonicMQ)	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). The following table lists the methods in the **Header** interface.

Table 18: Methods in the Header Interface (Continued)

Method	Description
void setContentId(java.lang.String cid)	Sets the ContentId header of the message or attachment part to the value cid.
setContentType(java.lang.String type)	Sets the ContentType header of the message or attachment part to the value type
void setHeaderField(java.lang.String name, java.lang.String value)	Sets the value of a header field name to the String value
java.lang.String getContentId()	Gets the content ID of the message or attachment part
java.lang.String getContentType()	Gets the ContentType of the message or attachment part. Returns the ContentType of the part or null
java.util.Enumeration getHeaderFieldNames()	Gets the list of all header fields
java.lang.String getHeaderField(java.lang.String name)	Gets the value of header field name or null if it does not exist
java.lang.String getHeaderField(java.lang.String name, java.lang.String value)	Gets the value of a header field name. If it does not exist default to value

Method	Description
<code>void removeHeaderField(java.lang.String name)</code>	Removes header name from the part
<code>void removeAllHeaders()</code>	Removes all headers

Using Multipart Messages to Wrap Problem Messages

A **MultipartMessage** is an efficient to handle undeliverable and indoubt messages even after they have been relegated to a Dead Message Queue (DMQ). You can create applications that screen DMQs to package lost messages and send them to a queue where they can be unpacked and analyzed.

Wrapping a Problem SonicMQ Message Within a Message

In this example, a **MessageListener** has received a message that it cannot handle and sends the message to an special-case application listening on **SpecialQ**.

To wrap a problem message and provide a reason code:

1. Wrap the entire original message as the payload for a **MultipartMessage** without changing the message body, headers, or properties.
2. Add an application reason code to the properties of the **MultipartMessage**.
3. Send the **MultipartMessage** to, in this example, **SpecialQ**.

The following code snippet shows an example of how to wrap a problem message.

Wrapping a Problem Message

```
void onMessage (Message m)
{
    // We only handle MapMessages
    if (m instanceof MapMessage)
        doNormalProcessing(m);
    else
    {
        // Send the message to the SpecialQ for processing
        MultipartMessage mm = session.createMultipartMessage();
        // Use JMS Properties on the mm to indicate an issue. DMQ these
        mm.setStringProperty("myError", "Can't handle this message");
        mm.setBooleanProperty(Constants.PRESERVE_UNDELIVERED, true);
        mm.setBooleanProperty(Constants.NOTIFY_UNDELIVERED, true);
        // Add the original message as the "Part"
        Part att = mm.createMessagePart(m);
        mm.addPart (att);
        specialSender.send(mm, PERSISTENT)
    }
}
```

Receiving a Wrapped Problem Message

In the preceding example, one SonicMQ message was wrapped inside another to encapsulate a problem. The following code snippet shows how that message is read and examined.

Receiving a Wrapped Problem Message (Continued)

```
// This is the MessageListener on the SpecialQ (for wrapped errors).
void onMessage (Message m)
{
    // ...
```

```

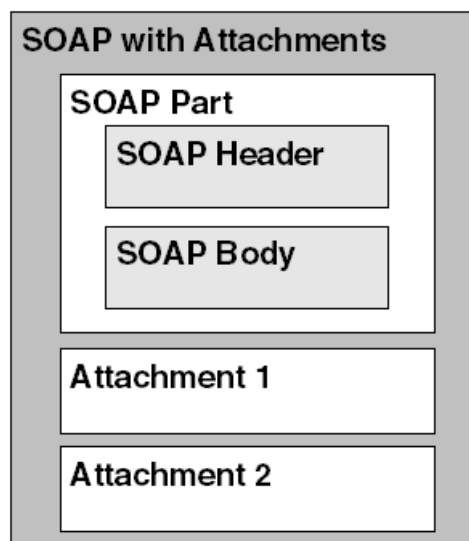
// Was it a normal error?
if (m instanceof MultipartMessage && m.getStringProperty("myError") != null)
{
    // This was a user error.
    log.out ("Error: " + m.getStringProperty("myError"));
    // Put out properties of the multipart message
    log.out("MessageID: " + m.getJMSMessageId());
    log.out("Destination: " + m.getJMSDestination());
    log.out("Mode: " + m.getJMSDeliveryMode());
    // Retrieve the original message(s)
    for (int i=0; i<m.getPartCount(); i++)
    {
        if (m.isMessagePart(i))
        {
            javax.jms.Message att = m.getPartAsMessage(i);
            log.out("\nPart # " + i);
            log.out("Original MessageID: " + att.getJMSMessageId());
            log.out("Original Dest: " + att.getJMSDestination());
            log.out("Original Mode: " + att.getJMSDeliveryMode());
        }
    }
}

```

Interacting with Business-to-Business Multipart Types

The SonicMQ **MultipartMessage** type supports B2B messaging in general and SOAP v.1.1. with Attachments in particular. SOAP message types are not expected to be handled at the JMS level. Instead, the application programmer wraps the underlying JMS **MultipartMessage** in a wrapper that implements the appropriate message, as shown in the following figure.

Figure 55: SOAP with Attachments as a MultipartMessage



This description shows how SonicMQ **MultipartMessages** coexist with business frameworks. Typically, the application looks at a JMS property and, based on the property value, creates an application object that uses or wraps the **MultipartMessage**. In the following example, the JMS header field **JMSType** holds the type of application object where it is intended to be used:

```

void onMessage (Message m)
{
    // Handle Multiparts by wrapping them
    if (m instanceof MultipartMessage)
    {
        MultipartMessage mm = (MultipartMessage) m;
        if (mm.getJMSType().equals("SOAP"))

```

```

{
// Use the SonicSW MessageFactory to expose this as a SOAP Message.
MessageFactory mf = new com.sonicsw.xc.MessageFactory ();
javax.xml.soap.Message soapm = mf.createSoapMessage( mm );

// Now we can do SOAP stuff on "soapm";
soapm.getSOAPPart();
...
}
...
}

```

Message Structure

JMS Messages are composed of the following parts:

- **Header Fields (JMS)** — 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** — JMS defines several types of message bodies, which cover the majority of messaging styles currently in use.

Note: While the JMS 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 the following table.

Table 19: Message Header Fields (Continued)

JMS Header Field	Type	Description	Usage	Comments
JMSDestination Required - Set by the producer send/publish method.	String	The destination where the message is being 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_REPLICATED, NON_PERSISTENT_SYNC, NON_PERSISTENT_ASYNC, or DISCARDABLE.	Default value is NON_PERSISTENT.
JMSMessageID Required - Set by the producer 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 send/publish method.	long	GMT time on the producer system clock when the message was sent.		Set method exists but is always overridden by the send method valuation.
JMSCorrelationID Optional - Set by producer method.	String	Broker-specified message ID or an application-specific String.	Required when other JMS providers support the native concept of a correlation ID.	An application made up of several clients might want an application-specific value for linking messages.
JMSCorrelationIDAsBytes 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.

JMS Header Field	Type	Description	Usage	Comments
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.
JMSTypeOptional - 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.
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; however, the JMS specification does not guarantee that this will not happen.	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 zero, 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.
JMSPriorityRequired - Set in a producer send/publish parameter.	int	Sets a value that will allow 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.	The JMS specification does not require that SonicMQ strictly implement priority ordering of messages; however, the broker will do its best to deliver expedited messages ahead of normal messages.

The basic method for producing a message allows essential delivery information to accept the JMS default values. For example:

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

Three of the message header fields have default values as static final 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 final int MESSAGE_LIFESPAN = 1800000;  
// milliseconds (30 minutes)  
sender.send( msg, javax.jms.DeliveryMode.PERSISTENT,  
javax.jms.Message.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.

Although the JMS specification does not define a policy for what should or should not be made a property, application developers should note that 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**. A **String** property is limited to 65,535 characters.

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.

The following table lists SonicMQ-defined properties.

Table 20: SonicMQ Provider-defined Properties

Function	JMS Provider-defined Property	Type	Set by
QoS setting	JMS_SonicMQ_perMessageEncryption	boolean	Producer
Message type	JMS_SonicMQ_Extended_Type	String	Producer
Handling of undeliverable messages	JMS_SonicMQ_preserveUndelivered	boolean	Producer
	JMS_SonicMQ_notifyUndelivered	boolean	Producer
	JMS_SonicMQ_undeliveredReasonCode	int	Broker
	JMS_SonicMQ_undeliveredTimestamp	long	Broker
	JMS_SonicMQ_destinationUndelivered	String	Broker
	JMS_SonicMQ_undeliveredBrokerName	String	Broker
	JMS_SonicMQ_undeliveredNodeName	String	Broker
	JMS_SonicMQ_undeliveredReasonAddedToDMQ	int	Broker
	JMS_SonicMQ_undeliveredOriginalJMSTDestination	String	Broker
	JMS_SonicMQ_undeliveredOriginalJMSTimestamp	long	Broker
	JMS_SonicMQ_undeliveredOriginalJMSEExpiration	long	Broker

Review the sample [Persistent Storage Application \(PTP\)](#) to see how the first properties are used. See [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 (QoS) settings on a security-enabled broker so that a client application that intends to produce to a destination must make the effort to send the message after the encryption and integrity requested has been performed. The client application is not aware of the QoS 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 QoS 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 (progress.message.jclient.Constants.ENCRYPT_MESSAGE, true);
```

You can determine whether a broker is security enabled by calling the **progress.message.jclient.Connection.isSecure()** method:

```
if (connect.isSecure())
{ aMessage.setBooleanProperty (progress.message.jclient.Constants.ENCRYPT_MESSAGE, true);
}
else
{
```

```
//Handle condition where broker is insecure...  
}
```

JMS-defined Properties (JMSX)

The JMS specification reserves the **JMSX** property name prefix 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. The following table lists and describes the JMSX Message Properties used in SonicMQ.

Table 21: JMSX Properties Used in SonicMQ (Continued)

JMSX Property	Type	Set by
JMSXGroupID	String	Producer on send
JMSXGroupSeq	int	Producer on send
JMSXUserID	String	Broker
JMSXDeliveryCount	int	Producer on receive

For more information about queue message grouping where the default grouping property is **JMSXGroupID**, and the **JMSXGroupSeq** is used to close a group assignment, see:

- The producer and consumer information in the [Using Message Grouping](#) .
- The broker settings on queues that handle message grouping, see the “Configuring Queues” chapter of the *Aurea SonicMQ Configuration and Management Guide*.

For more information about setting the sending user name of the message in **JMSXUserID**:

- As used in basic authentication with HTTP Direct, see the “HTTP(S) Direct Acceptors and Routings” chapter of the *Aurea SonicMQ Deployment Guide*.
- As a brokerwide setting for TCP and SSL connections, see advanced broker property settings in the “Configuring SonicMQ Brokers” chapter of the *Aurea SonicMQ Configuration and Management Guide*.

For more information about **JMSXDeliveryCount**:

- As used to programmatically handle excessive redeliveries, see [Setting Maximum Delivery Count](#) .
- As used to automatically handle excessive redeliveries, see [Setting Maximum Delivery Count](#) , and the “JMS Administered Objects Tool” chapter in the *Aurea SonicMQ Configuration and Management Guide*.

User-defined Properties

A message supports application-defined property values, providing a mechanism for adding application-specific header fields to a message. For examples:

- Identifiers for audits or reconciliation of undeliverable messages. These might be properties you define such as **OriginatorHostID**, **AuditID**, **RealTimeDeviceID**, **RFID**, or similar.
- Hints for rerouting undeliverables such as **ReturnURL**, **AlternateURL**, **ReturnDestination**, **ReturnEmail**, or similar.
- Settings that are 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 *Aurea SonicMQ Deployment Guide*. Examples of such properties are:
 - X-HTTP-AuthUser and **X-HTTP-AuthPassword** for Web server authentication.
 - **X-HTTP-ReplyAsSOAP**, **X-HTTP-RequestTimeout**, **X-HTTP-Retries**, and **X-HTTP-RetryInterval** These 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 JMS 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 destination that was created as a node-qualified HTTP destination URL to group messages for the same destination, sending them through the same pending queue after normalizing the URL into patterns.

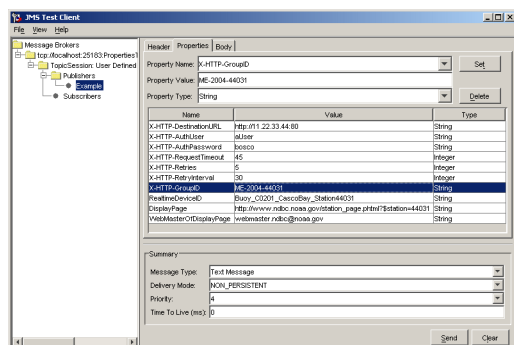
For more information, see the "Grouping Messages by Destination URL" section of the "HTTP(S) Direct Acceptors and Routings" chapter in the *Aurea SonicMQ Deployment Guide*.

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 *Aurea 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.

An example of some custom properties for HTTP outbound and attached custom information is shown in the following figure captured in a JMS Test Client session.



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. 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.

While JMS-defined properties are typed, user-defined properties are not. 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.

Checking Whether a Property Exists

Use the **propertyExists()** method to check whether a property value exists:

```
public boolean propertyExists(String name)
```

wherename 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 have been 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 the following table 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 22: Permitted Type Conversions for Message Properties (Continued)

	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**; those intersections marked with **No** throw a **JMSEException**. 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 message **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 specified by JMS for all types except XML unless the message is read-only (in which case you will need to 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 **XMLMessage** body, see [Working with XML Messages](#) on page 194 and the samples XML Messages.

For information about setting the **Parts** into a **MultipartMessage**, see [Composition of a MultipartMessage](#) on page 197.

Getting the Message Body

Use the **get()** methods required by the JMS specification for all types except XML. For example:

```
msg.getText (aMessage) ;
```

For information about getting **XMLMessage** body and interpreting it with DOM or SAX parsers, see [Working with XML Messages](#) on page 194 and the samples XML Messages.

For information about getting **Parts** of a **MultipartMessage** and distinguishing JMS message types from other MIME types, see [MultipartMessage Type](#) on page 198 and the sample [Decomposing Multipart Messages](#) .

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 [Point-to-point Messaging](#) and [Publish and Subscribe Messaging](#) .

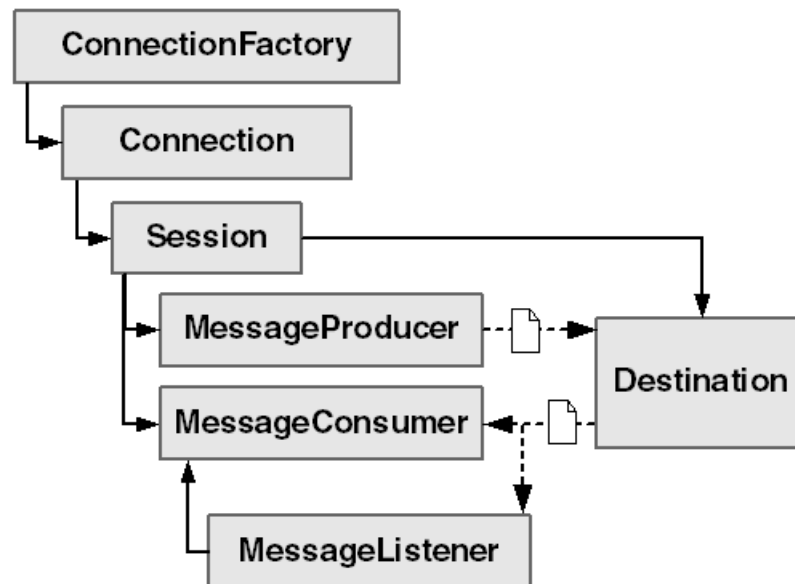
For details, see the following topics:

- [About Message Producers and Message Consumers](#)
- [Message Ordering and Reliability](#)
- [Destinations](#)
- [Steps in Message Production](#)
- [Message Management by the Broker](#)
- [Message Receivers, Listeners, and Selectors](#)
- [Steps in Listening, Receiving, and Consuming Messages](#)
- [Reply-to Mechanisms](#)
- [Producers and Consumers in JMS Messaging Domains](#)

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 the following figure.

Figure 56: 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](#) and [Message Ordering and Reliability in Pub/Sub](#) 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** will still exist. Ordering is maintained while expiration is a user-defined value.
- The producer can set the **delivery mode** of messages so that 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 take efforts to 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 it is 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** technique—transactional or not, explicit or implicit.
- Connections can be monitored and, when broken, techniques can automatically attempt to **reconnect**. (This might not be necessary if you are using fault-tolerant connections. See [Fault-Tolerant Connections](#).)
- 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 JMS Administered Objects (static objects under administrative control), dynamic objects created as needed (topics only), or temporary objects created for very limited use. The destination name length limit is 256 characters.

For topics, SonicMQ provides extended management and security with **hierarchical name spaces**; for example, **jms.samples.chat**. See [Hierarchical Name Spaces](#) for more information.

Important: [Connection Factories](#) on page 117 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 *Aurea SonicMQ Configuration and Management Guide* for more information.
- 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 using the Administration Shell.

You can programmatically store and retrieve defined destinations. SonicMQ lets you store topic or queue names in JNDI or a simple file store and then reference the object indirectly (by name) in some context. See [Lookup and Use of Administered Objects](#) for more information.

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**. These sections follow the approach used in the **Chat** sample:

1. [Create a Session](#) on page 218
2. [Create the Producer on the Session](#) on page 219
3. [Create the Message Type and Set Its Body](#) on page 219
4. [Set Message Header Fields](#) on page 219
5. [Set the Message Properties](#) on page 220
6. [Elect Per Message Encryption](#) on page 220 (optional)
7. [Produce the Message](#) on page 220

Create a Session

After establishing a connection, the Chat sample creates a **Session**:

```
javax.jms.Session pubSession;  
...  
pubSession = connect.createSession(false, javax.jms.Session.AUTO_ACKNOWLEDGE);
```

You can use a **Session** for P2P 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 static variable **APP_TOPIC** (assigned the value “`jms.samples.chat`”) as the working **Topic** and creates a **MessageProducer** associated with that **Topic**:

```
private static final String APP_TOPIC = "jms.samples.chat";
...
private javax.jms.MessageProducer publisher = null;
...
javax.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:

```
String s = stdin.readLine();
javax.jms.TextMessage msg = pubSession.createTextMessage();
msg.setText( username + ": " + s );
```

When the sample is run, if the user **Sales** enters “**Hello.**”, the message content would be “**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(progress.message.jclient.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**, as follows:

```
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, as follows:

```
private static final long MESSAGE_LIFESPAN = 1800000;
publisher.send(msg, javax.jms.DeliveryMode.PERSISTENT,
    javax.jms.Message.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**|**NON_PERSISTENT_REPLICATED**|**DISCARDABLE**]
- **JMSPriority** is [**0...9**] where **0** is lowest, **9** is highest, **4** is the default
- **timeToLive**, the message lifespan that will calculate 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 23 lists those parameters and how they direct the broker to handle the message.

Note: Asynchronous message delivery — Sonic's asynchronous message delivery is set on a connection factory to give a non-transacted session increased performance for delivery modes that are not explicitly asynchronous—**NON_PERSISTENT** on a security-disabled broker and **NON_PERSISTENT_ASYNC** delivery mode. This feature adds asynchronous operation to the **NON_PERSISTENT_REPLICATED** delivery mode, a delivery mode used by fault-tolerant brokers replicating nonpersistent messages from the active peer to its standby. See the section "Asynchronous Message Delivery" in the "SonicMQ Connections" chapter of the *Aurea SonicMQ Application Programming Guide* for detailed information about this connection factory setting and its associated behaviors.

Table 23: How Message Producer Parameters Influence the Broker

Producer Parameter	How the parameter is treated by the broker
deliveryMode	<p>deliveryMode = PERSISTENT — Stores the message in the broker's message log in case of impending failure. Acknowledges the producer only after logging the message.</p> <p>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.</p> <p>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. Also, some exceptions that might otherwise be thrown back to the client when it sends a message are not communicated; for example, a message that is larger than the queue size could seem to be a lost as the client did not get the exception and then fail or crash.</p> <p>deliveryMode = NON_PERSISTENT_SYNC — This is the default nonpersistent delivery mode when security is enabled. Message publisher methods block to await acknowledgement.</p> <p>deliveryMode = NON_PERSISTENT_REPLICATED — Used with fault-tolerant connections. In this mode, non-persistent messages are protected from broker failures by being replicated to a standby broker. This delivery mode also ensures once-and-only-once delivery to fault-tolerant subscribers (both durable and non-durable) provided that after a failure the subscriber either successfully resumes its connection at the same broker or fails over to the standby broker when that broker takes the active role.</p> <p>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</p> <p>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.</p>
priority	<p>priority = 0...9</p> <p>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.</p>
timeToLive	<p>timeToLive = <non-negative long integer value></p> <p>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, the indication that the message is intended 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.</p>

Message Receivers, Listeners, and Selectors

MessageConsumer objects that are associated with a **Topic** do not automatically get messages. Having an active session where an application subscribes to a topic does not result in the message getting 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, use the **receive()** method with a timeout:

```
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, use the **receiveNoWait()** method:

```
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 paradigm. 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 paradigm where messages wait on a static queue.

Message Listeners

Invoke a message listener to initiate asynchronous monitoring of the session thread for consumer messages by using the following method:

```
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 that the listener is bound to the destination to which a consumer was just created. For example, in PTP:

```
javax.jms.MessageConsumer receiver = session.createConsumer(queue,  
java.lang.String messageSelector);  
receiver.setMessageListener(this);
```

Another example, this time in Pub/Sub:

```
javax.jms.MessageConsumer subscriber = session.createConsumer(topic,  
java.lang.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 get every message produced to a destination, other applications might want to receive only certain messages produced to a destination. The following techniques that 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 [Hierarchical Name Spaces](#).
- **Applying a message selector** — As shown in the preceding code examples, JMS 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 its 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](#) 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, JMS message selectors do not. SQL comments are not supported.

A selector String greater than 1024 characters will throw an exception.

Message Selector Syntax

A message selector is a **java.lang.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**
- **Operators and Expressions**
- **Comparison tests**
- **Parentheses** control the evaluation of an expression
- **Whitespace** (spaces, horizontal tabs, form feeds, and line terminators) are evaluated in the same way as in Java

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 24: Literal and Identifier Syntax in Message Selectors (Continued)

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

Selector	Element	Format and Requirements	Constraints	Example
Identifiers	All	A case-sensitive character sequence that must begin with a Java-identifier start character. All following characters must be Java-identifier part characters. A Java-identifier is an unlimited-length sequence of Java letters, Java digits, and, “for historical reasons,” the underscore (_) and dollar sign (\$) characters. The first character of a Java-identifier must be a Java letter. For more about Java-identifiers, see the Java Language Specification’s Lexical Structure chapter at java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.8	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			JMSXState JMSXState Undelivered
	Application-specific property names (do not start with ‘JMS’)			Audit_Team

Table 25: Operator and Expression Syntax in Message Selectors (Continued)

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 * 3a - 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

Selector	Element	Format and Requirements	Example
Expressions	Selector	Conditional expression that matches when it evaluates to true	$((4*3)=(2*6))= \text{true}$
	Arithmetic	Include: Pure arithmetic expressions Arithmetic operations Identifiers with numeric values Numeric literals	$7*5a/b7$
	Conditional	Include: Pure conditional expressions Comparison operations Logical operations Identifiers with Boolean values Boolean literals (true, false)	$7>6a > 7 \text{ OR } b = \text{true } a = \text{true}$

Table 26: Comparison Test Syntax in Message Selectors (Continued)

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 Esc character.	a LIKE 'Fr%d' is true for 'Fred' 'Frond' and false for 'Fern' a LIKE '_%' ESCAPE '\ ' true for '_foo' and false for 'bar'
	null	Identifier IS NULL Identifier IS NOT NULL for: Header field value Property value Existence of a property Refer to SQL-92 semantics or the JMS specification for more about comparisons that involve null values.	a is NULL a is NOT NULL

Comparing Exact and Inexact Values

Comparing an **int** value (an exact numeric literal that uses the Java integer literal syntax) and a **float** value (an approximate literal that uses the Java floating point literal syntax) is allowed.

Type conversion is defined by the rules of Java numeric promotion as described in the Java Language Specification, which, in part, declares that:

- Unary conversions are from **byte**, **short**, or **char** to a value of type **int** by a widening conversion; otherwise, a unary numeric operand remains as is and is not converted.
- Binary conversions 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 228
2. [Create the Destination and Consumer, Then Listen](#) on page 228
3. [Handle a Received Message](#) on page 228
 - a. [Get Message Properties](#) on page 229
 - b. [Consume the Message](#) on page 229
 - c. [Acknowledge the Message](#) on page 230

Implement the Message Listener

Implement the standard JMS message listener:

```
public class Chat
    implements javax.jms.MessageListener
...

```

Create the Destination and Consumer, Then Listen

Once you obtain a **ConnectionFactory** object, use it to create a **Connection**. From the **Connection**, create a **Session**, and, from the **Session**, 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 P2P 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**):

```
javax.jms.Topic topic = subSession.createTopic("jms.samples.chat");
javax.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 void onMessage( javax.jms.Message aMessage )
{
    javax.jms.TextMessage textMessage = (javax.jms.TextMessage) aMessage;
    String string = textMessage.getText();
    System.out.println( string );
}

```

When the received message type is uncertain, special message handling is required. In the following **XMLDOMChat** sample, the message is tested to determine whether or not it is an instance of **XMLMessage** and then handled appropriately:

```
public void onMessage( javax.jms.Message aMessage )
{ if (aMessage instanceof progress.message.jclient.XMLMessage)
  { ... see Parsing an XML Message }else{
    // Cast the message as a text message and display it.
    javax.jms.TextMessage textMessage = (javax.jms.TextMessage) aMessage;
    System.out.println( "[TextMessage] " + textMessage.getText());
  }
}
```

When the received message is an XML message, your application can parse the message to extract data from the message fields. The following code sample shows how to parse an XML message and extract data:

```
// Cast the message as an XML message.
progress.message.jclient.XMLMessage xmlMessage =
(progress.message.jclient.XMLMessage) aMessage;
//Get the XML document associated with this message.
org.w3c.dom.Document doc = xmlMessage.getDocument();
// Get the sender and content from the message.
org.w3c.dom.NodeList nodes = null;
nodes = doc.getElementsByTagName("sender");
String sender = (nodes.getLength()> 0)
nodes.item(0).getFirstChild().getNodeValue() : "unknown";
nodes = doc.getElementsByTagName("content");
String content = (nodes.getLength()> 0) ?
nodes.item(0).getFirstChild().getNodeValue() : null;
//Show the message.
System.out.println("[XML from '" + sender + "'" + content);
// Show the message as a tree.
printDocNodes(doc.getDocumentElement(),0);
System.out.println();
```

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 will force coercion of the value to the declared type. If the conversion is not valid, an exception is thrown. See [Getting Property Values](#) on page 213.

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 JMS message to the broker could be postponed until the business application acknowledges processing with a transaction or audit trail identifier. This value could be passed back to the producer if a reply was 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 ack acknowledges all messages previously received by the session.

Note: The **acknowledge** method has no effect when the session is transacted or when the session ack mode is **AUTO_ACKNOWLEDGE** or **DUPS_OK_ACKNOWLEDGE**. See [Explicit Acknowledgement](#) 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** would typically contain 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. 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**:

```
        javax.jms.TopicRequestor requestor = new javax.jms.TopicRequestor(session,
topic);
        javax.jms.Message response = requestor.request(msg);
        javax.jms.TextMessage textMessage = (javax.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:

```
//get the message
public void onMessage( javax.jms.Message aMessage)
{ javax.jms.TextMessage textMessage = (javax.jms.TextMessage) aMessage;
  String string = textMessage.getText();
}
...
//Look for the header specifying JMSReplyTo:
javax.jms.Topic replyTopic = (javax.jms.Topic) aMessage.getJMSReplyTo();
if (replyTopic != null)
...
//Send a reply to the topic specified in JMSReplyTo:
javax.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 by the design paradigm for JMS requests:

- Set the **JMSCorrelationID**, tying the response back to the original request.
- Use transacted session **commit** so that the request will not be received without the reply being sent. For example:

```
reply.setJMSCorrelationID(aMessage.getJMSMessageID());
replier.send(replyTopic, reply);
session.commit();
```

Writing a Topic Requestor

The default **TopicRequestor** behavior is to block when waiting for a reply. You can write your own **TopicRequestor** class that will timeout (**receive(long timeout)**) or listen to the temp topic as a subscriber, thereby avoiding the blocking situation. The **javax.jms.TopicRequestor.java** file, shown in [Writing a Topic Requestor](#) on page 231, is a start toward creating your own **TopicRequestor.class**.

Writing a Topic Requestor

```
// @(#)TopicRequestor.java 1.9 98/07/08
// Copyright (c) 1997-1998 Sun Microsystems, Inc. All Rights Reserved.
package javax.jms;
public class TopicRequestor
// The topic session the topic belongs to.
```

```
{ TopicSession    session;
// The topic to perform the request/reply on.
Topic topic;

// Constructor for the TopicRequestor class.
TemporaryTopic    tempTopic;
TopicPublisher    publisher;
TopicSubscriber    subscriber;
public TopicRequestor(TopicSession session, Topic topic) throws JMSEException
{
    this.session = session;
    this.topic    = topic;
    tempTopic     = session.createTemporaryTopic();
    publisher     = session.createPublisher(topic);
    subscriber    = session.createSubscriber(tempTopic);
}

// Send a request and wait for a reply.
public Message request(Message message) throws JMSEException
{
    message.setJMSReplyTo(tempTopic);
    publisher.publish(message);
    return(subscriber.receive());
}

// Close resources when done.
public void close() throws JMSEException
{
    tempTopic.delete();
    publisher.close();
    subscriber.close();
    session.close();
}
}
```

Producers and Consumers in JMS Messaging Domains

The following table lists a general messaging functionality that is consistent in both Publish and Subscribe and Point-to-point messaging.

Table 27: Producer and Consumer Common to Both Messaging Models

javax.jms Interface	Functionality in Either Domain
Destination Extended by: Queue, Topic	Destination supports concurrent use
MessageProducer	<ul style="list-style-type: none"> • Able to send message while connection is stopped • Close MessageProducer method • Supports message delivery modes PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_SYNC, NON_PERSISTENT_ASYNC, NON_PERSISTENT_REPLICATED, and, for topics only, DISCARDABLE • Supports message Time-to-Live • Support message priority
MessageConsumer	<ul style="list-style-type: none"> • 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 and TextMessage Extended by: Message, TextMessage, BytesMessage, MultipartMessage	<ul style="list-style-type: none"> • Message header fields • Message properties • Message acknowledgment • Message selectors • Access to message after being sent for reuse

See [Point-to-point Messaging](#) and [Publish and Subscribe Messaging](#) for programming concepts and functionality in each messaging domain.

Point-to-point Messaging

This chapter describes the Point-to-point (PTP) messaging model and explains how to use the features of that model.

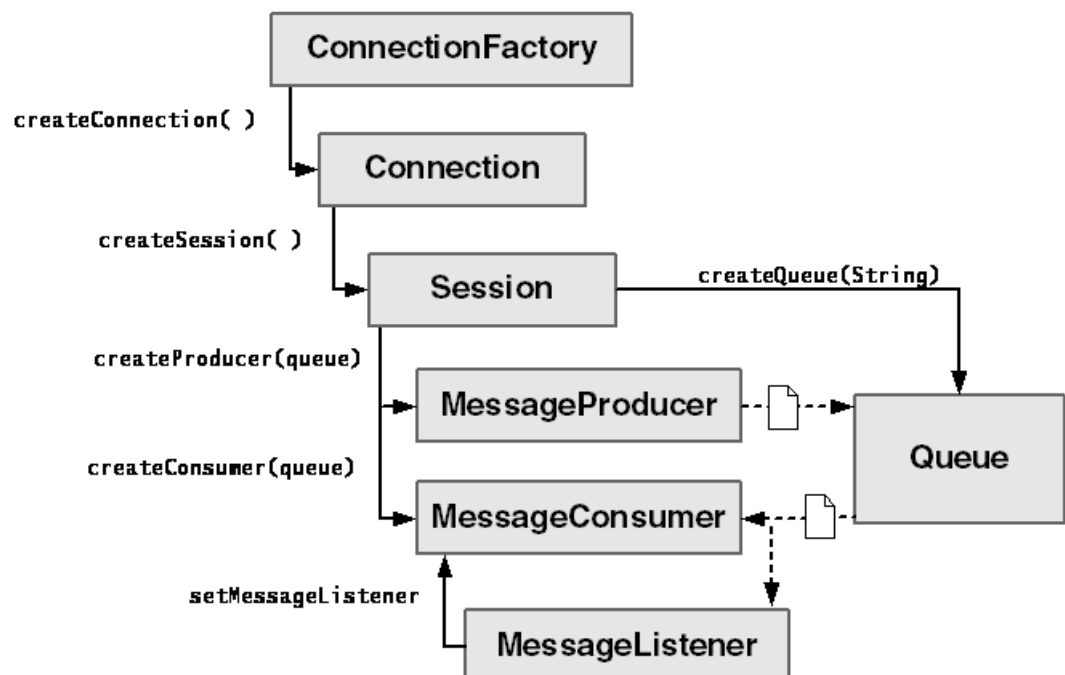
For details, see the following topics:

- [About Point-to-point Messaging](#)
- [Message Ordering and Reliability in PTP](#)
- [Using Multiple MessageConsumers](#)
- [Using Message Grouping](#)
- [Setting Prefetch Count and Threshold](#)
- [Browsing a Queue](#)
- [Handling Undelivered Messages](#)
- [Life Cycle of a Guaranteed Message](#)
- [Detecting Duplicate Messages](#)
- [Forwarding Messages Reliably](#)
- [Dynamic Routing with PTP Messaging](#)
- [Clusterwide Access to Queues](#)

About Point-to-point Messaging

In the Point-to-point (PTP) messaging model, shown in the following figure, a queue stores messages for as long as they are specified to live, waiting for a consumer. The **QueueBrowser** class enables an application to browse the queue, examine its contents, and observe how message traffic is moving.

Figure 57: Point-to-point Messaging Model



Queues must be created by the administrator before they can be used (except for temporary queues, which are created dynamically by users). See the *Aurea® 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 that will be used:

```
java 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.

The following code snippet, from the **Talk** sample, shows how to create the objects used in PTP communication.

Talk Sample: Creating Objects for PTP

```
// Create a connection. (try/catch)
javax.jms.ConnectionFactory factory;
factory = (new progress.message.jcclient.ConnectionFactory (broker));
connect = factory.createConnection (username, password);
sendSession = connect.createSession (false, javax.jms.Session.AUTO_ACKNOWLEDGE);

receiveSession = connect.createSession (false, javax.jms.Session.AUTO_ACKNOWLEDGE);

// Create Sender and Receiver 'Talk' queues. (try/catch)
if (sQueue != null)
{
    javax.jms.Queue sendQueue = sendSession.createQueue (sQueue);
    sender = sendSession.createProducer (sendQueue);

    if (rQueue != null)
    {
        javax.jms.Queue receiveQueue = receiveSession.createQueue (rQueue);
        javax.jms.MessageConsumer qReceiver = receiveSession.createConsumer(receiveQueue);

        qReceiver.setMessageListener(this);
        // The 'receive' setup is complete. Start the Connection
        connect.start();
        ...
    }
}
```

Message Ordering and Reliability in PTP

The PTP messaging model has unique features in message ordering and delivery.

Message Ordering

Queued delivery allows each message to be processed by one and only one message consumers. 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 factors that impact the ordering and reliability of messages:

- When a new message is put onto a queue with a high **priority** set by the sender, an active message consumer takes the new message off the queue before taking an older message with 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 have been 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 will receive. 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 will clear the index value, but any intervening sends that specify an already recorded identifier are rejected.

Note: The effects of dynamic routing on message ordering and delivery are discussed at greater length in the scenarios in the *Aurea 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** will only receive 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 will effectively generate 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 will be invoked when a message is delivered.

Message Queue Listener

A message listener is used to allow asynchronous processing of queue messages:

```
setMessageListener(MessageListener listener)
```

where **listener** is the message listener to associate with this session.

The listener is often assigned just after creating the consumer from the session, as shown:

```
javax.jms.Queue receiveQueue = session.createQueue (rQueue);  
javax.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.

Using Message Grouping

When there are multiple receivers on a queue, messages are distributed round-robin to the active receivers. That could mean that the messages for a group (such as a customer or stock ticker symbol) could get recorded out of order by the receiver applications.

To avoid this behavior, you could create an exclusive queue for each group so that only one receiver gets those messages in strict order. Or, you could define message selectors for a receivers that each select one category of messages. But in both cases, you need to set up a known, static set of producer destinations, queues, and consumers.

Message grouping—defined by the message producers sending to specifically configured queues on the broker—provides a mechanism that you might think of as dynamic exclusive subqueues.

The producer sets a property with a message's group value, and sends the message to the broker. The broker determines whether it has an assigned active receiver for the message's group:

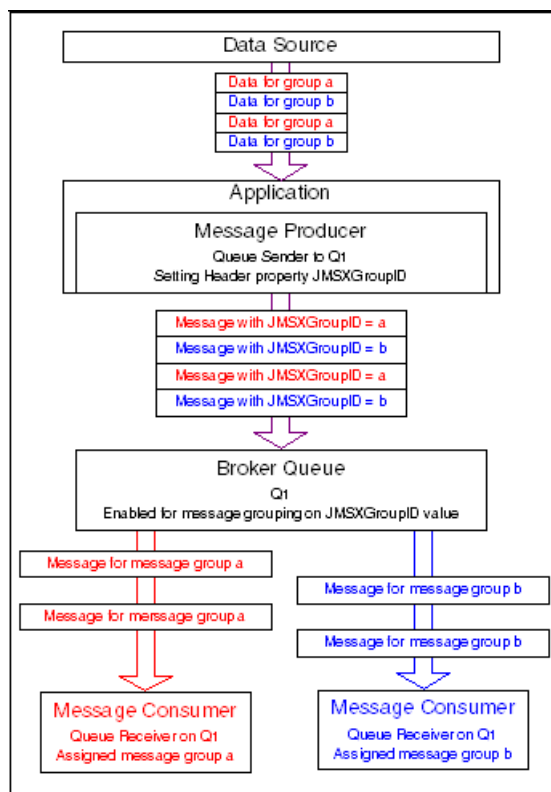
- If it does, the message goes to the assigned receiver at the end of undelivered messages.
- If it does not, it assigns a receiver to the group name.

This feature provides advantages when the incoming messages are for diverse groups with arbitrary names that handle a reasonable number of messages. The limitation is that, like exclusive queue receivers, when the backlog on a message group's receiver gets huge, you cannot add receivers to balance the load.

Note: MessageGroupTalk sample application — Try the message grouping sample [MessageGroupTalk \(PTP\)](#) .

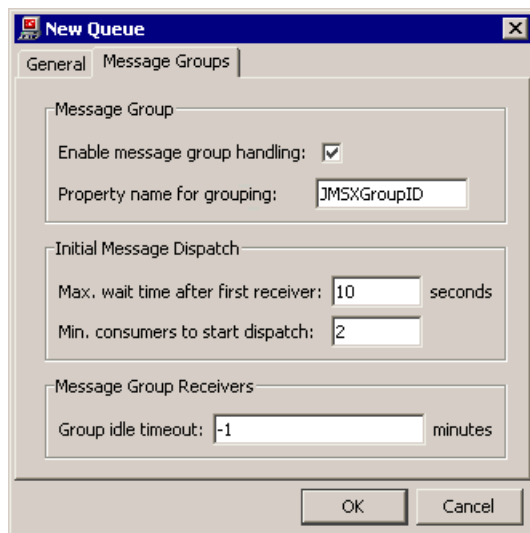
Illustration of Message Grouping

The following illustration shows how source data sent for two groups (**a** and **b**) is set on the message grouping property of each message from a message producer. The producer connects to a queue that is enabled for message grouping. The two receivers on the queue are each assigned a message group so subsequent messages with the same group identifier are channeled to the assigned group receiver.



Broker Settings for Message Grouping

A SonicMQ broker must enable message grouping on a per-queue basis before message grouping can occur. Each queue configuration enables message group dispatching of messages to receivers and specifies the property that will determine the grouping of messages. The default property is **JMSXGroupID**.



Other queue properties set on the broker help it control the message groups.

Initial Message Dispatch

Manages the assignment of messages and message groups when receivers start:

- **Max. wait time after first receiver** — Specifies a delay after the first consumer is available to avoid forcing all the message groups that receiver. The default value is 10 seconds.
- **Min. consumers to start dispatch** — Specifies the number of consumers that, when receiving on the queue, will allow dispatch of messages before the wait time has elapsed. The default value is 2. If the **Max wait time** is 0, this setting has no effect.

Group Idle Timeout

Once groups are established, the broker needs overhead to keep track of assigned group receivers. You can set a timeout for a group assignment that has not received any messages in the elapsed time. Subsequent messages to the group will be assigned again, possibly to the same receiver that was timed out.

Message Producers for Message Grouping

A producer application controls a message's group handling with two **StringProperty** settings, the group identifier and the group sequence.

Creating and Sending to a Message Group

The message grouping property name on a message sent to queue to enabled for message grouping to perform message group dispatching to receivers. The value of that property will be a group, even if the group value is "".

```
Message message = session.createTextMessage("Order for customer Acme");
message.setStringProperty("JMSXGroupID", "ACME");
...
producer.send(message);
```

If the message grouping property is not on a message, it is distributed in the standard round-robin to active receivers.

Requesting the Broker to Unassign a Message Group

The producer can set a demarcation (for example, end of order or end of day) in a stream of messages to a message group by specifying the group's value in the message group property, and also set the value **JMSXGroupSeq** to the value 0. For example:

```
Message message = session.createTextMessage("EOF");
message.setStringProperty("JMSXGroupID", "PRGS");
message.setIntProperty("JMSXGroupSeq", 0);
...
producer.send(message);
```

The broker would no longer deliver new messages to the assigned receiver for the group, and any subsequent messages for that group name would get re-assigned (possibly to the same receiver as before.)

For scenarios where the producer demarcation may be used to signal to the consumer that it should commit its work before the group is re-assigned, use the value -1 instead of 0 to instruct the broker to close the group, but to wait until the last message of the group has been acknowledged before doing so.

Note: You can, at your discretion, use **JMSXGroupSeq** positive integer values to iterate the messages sent to a group in producer and receiver applications.

Message Consumers for Message Grouping

Receivers on a queue enabled for message grouping are managed by the broker such that:

- A message for a group name that is not currently assigned to a receiver is assigned to an active receiver selected through a round robin algorithm.
- A message for a group name that is currently assigned to a receiver is delivered to the assigned receiver. A receiver that is assigned a message group will be the only receiver of messages for that group as long as the receiver is active and the producer has not explicitly called for the assignment to end. When a receiver closes, it forsakes its assigned message groups; the unacknowledged messages for those groups create corresponding message group assignments to other active receivers.
- A message without the group property specified is assigned to an active receiver through a round robin algorithm.

Receivers on a queue that is enabled and performing message grouping are not aware of their assigned groups. Message group assignments are made and managed by the broker.

If message grouping is used with JMS Connection Consumers, for example in the context of an application server, the connection consumer is the receiver. All messages for a given message group are delivered to the same connection consumer, but from there messages are handed off to any available session in the connection consumer's session pool. Consequently, there's no guarantee that all messages in a group are handled by the same application server thread or Message Driven Bean instance. This is true even if the application server does not use connection consumers, unless there's a static one-to-one mapping between JMS receivers and MDB instances, which typically would not be the case.

Important: When message group handling is enabled on a queue, message selectors on that queue are not valid. Applications that attempt to receive messages on a queue that is enabled for message grouping will fail if the application uses message selectors.

Message group assignments endure across failover to a standby broker as long as the fault-tolerant receivers re-establish connections before timing out.

Setting Prefetch Count and Threshold

SonicMQ extends the standard **MessageConsumer** interface, enabling you to set and get the following parameters of the message receiver that allow performance tuning:

- **PrefetchCount** — The number of messages that the consumer will take 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:`
`progress.message.jclient.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. This can improve performance.

- `getPrefetchCount:`
`progress.message.jclient.MessageConsumer.getPrefetchCount()`

Returns the **PrefetchCount** positive integer value.

Use the following **set()** and **get()** methods for the prefetch threshold:

- `setPrefetchThreshold:`
`progress.message.jclient.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 will be fetched.

- `getPrefetchThreshold:`
`progress.message.jclient.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 may 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](#) 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: **java.util.Enumeration 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()**

The sample application **QueuePTP\QueueMonitor** uses the **QueueBrowser** to display current queue contents in a Java window, as shown in the following code snippet.

QueueBrowser Sample

```
// Create a browser on the queue and show the messages waiting in it.
javax.jms.Queue q = (javax.jms.Queue) theQueues.elementAt(i);
textArea.append("Browsing queue \"" + q.getQueueName() + "\"\n");

// Create a queue browser
System.out.print ("Creating QueueBrowser for \"" + q.getQueueName() + "\"...");

javax.jms.QueueBrowser browser = session.createBrowser(q);
System.out.println ("[done]");
int cnt = 0;
Enumeration e = browser.getEnumeration();
if(!e.hasMoreElements())
{ textArea.append("<no messages in queue>");
}
else
{
    System.out.print (" --> getting message " + String.valueOf(++cnt) + "...");
    javax.jms.Message message = (javax.jms.Message) e.nextElement();
    System.out.println("[ " + message + " ]");
    if (message != null)
    {
        String msgText = getContents (message);
        textArea.append(msgText + "\n");
        ...
    }
    ...
}
...
}
```

See [Browsing Clusterwide Queues](#) on page 254 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**.

As a programmer, you can elect to request that undeliverable messages be 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 *Aurea 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. The following code snippet shows how to set messages for a **PERSISTENT** delivery mode to be preserved if undeliverable and to generate a notification if undeliverable.

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.
javax.jms.BytesMessage msg = session.createBytesMessage();
msg.setBytes(payload);

// Set 'undelivered' behaviour.
msg.setBooleanProperty(PRESERVE_UNDELIVERED, true);
msg.setBooleanProperty(NOTIFY_UNDELIVERED, true);

// Send the message with PERSISTENT, TimeToLive values.
qsender.send(msg, javax.jms.DeliveryMode.PERSISTENT,
    javax.jms.Message.DEFAULT_PRIORITY, 7200000);
```

Setting Small Messages to Generate Administrative Notice

To determine if 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. The following code snippet shows how to set messages to generate administrative notice.

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.
javax.jms.BytesMessage msg = session.createBytesMessage();
msg.setBytes(payload);

// Set 'undelivered' behavior. Using the property names that
// are defined as static final Strings in
// progress.messages.jclient.Constants ensures catching errors.
msg.setBooleanProperty(NOTIFY_UNDELIVERED, true);

// Send the message for fast delivery, or not at all.
qsender.send(msg,
    javax.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 gets 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 [Guaranteeing Messages](#) for information about using the DMQ and guaranteeing messages.

Setting the Message to Be Preserved

The application developer 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(progress.message.jclient.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 dequeuing undelivered messages. To declare that an administrative event should be generated, set the appropriate message property:

```
msg.setBooleanProperty(progress.message.jclient.Constants.NOTIFY_UNDELIVERED,
    true);
```

Sending the Message

The sending application sends the message metadata and the message payload. The application can expect that the message gets 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 will be 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.

Dequeuing 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 JMS 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 will not expire. If you have access to administrative functions on a broker, stay alert and dequeue dead messages as soon as possible. Messages with **NON_PERSISTENT** delivery mode are volatile and will 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 was made 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 has been saved in the DMQ on the broker where the message was declared undeliverable.

Getting Messages Out of the Dead Message Queue

The following code snippet shows the use of synchronous receives for messages in the DMQ.

Getting Messages from the DMQ

```
import progress.message.jclient.Constants;
...

// Create a MessageConsumer for the dead message queue.
Session session = connect.createSession(false, Session.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(Constants.UNDELIVERED_REASON_CODE);
    if (code == 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 **program.message.jclient.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 will throw 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(java.lang.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 gets rolled back because a duplicate UUID is detected, the following exception is thrown: **TransactionRolledBackException**.

Note: Transactions using this commit feature will be slower than normal transactions.

For more information about duplicate messages, see [Duplicate Message Detection Overview](#) on page 288.

Forwarding Messages Reliably

The **progress.message.jclient.Message** class provides the **acknowledgeAndForward** method to reliably acknowledge a message received from a queue destination and to forward the message to another queue destination.

To acknowledge a message and forward it to a new destination, use the method:

```
public void acknowledgeAndForward (javax.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 **progress.message.jclient.Session.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 have been 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 JMS 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 would result in the use of the priority and delivery mode values of the message while the interval between the timestamp and the expiration time would be used as the time to live:

```
public void acknowledgeAndForward (javax.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 destination or message producer destination under **acknowledgeAndForward**.

The following procedure describes how you can modify one of the SonicMQ sample applications to demonstrate the **acknowledgeAndForward()** method.

Modifying a sample to show the acknowledgeAndForward behavior

To modify a sample to show the **acknowledgeAndForward** behavior:

1. Copy the **ReliableTalk** sample source file, `samples\QueuePTP\ReliableTalk.java`.
2. Change the line **textMessage.acknowledge()** to:

```
javax.jms.Queue sendQueue = sendSession.createQueue(m_sQueue);
((progress.message.jclient.Message) textMessage).acknowledgeAndForward(sendQueue);
```

3. Change the line **receiveSession = connect.createSession...** to:

```
receiveSession =
connect.createSession(false,progress.message.jclient.Session.SINGLE_MESSAGE_ACKNOWLEDGE);
```

4. Save the modified file and then compile it into a class 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. The Sonic Dynamic Routing Architecture (DRA) is based on a similar concept. Most of the DRA complexity is managed in the communication layer, so that 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

The SonicMQ DRA provides active route optimization and accelerated acknowledge-and-forward transactional message forwarding while minimizing programmatic overhead.

Administrative Requirements

In all cases of 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 *Aurea 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 variable 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 has been set to be global

The *Aurea 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 chapter “Multiple Nodes and Dynamic Routing” in the *Aurea 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 chapter “HTTP Direct Acceptors and Routings” in the *Aurea 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 was 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 the 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 239). 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 244). 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](#) and [Browsing a Queue](#) on page 244). A clustered queue instance on a broker will receive 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 will not need 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 will not be available for browsing or receiving until the unavailable broker is restarted.

Note: Clustered queues do not support the enforcement of strict message ordering.

Publish and Subscribe Messaging

This chapter describes the Publish and Subscribe (Pub/Sub) messaging model .

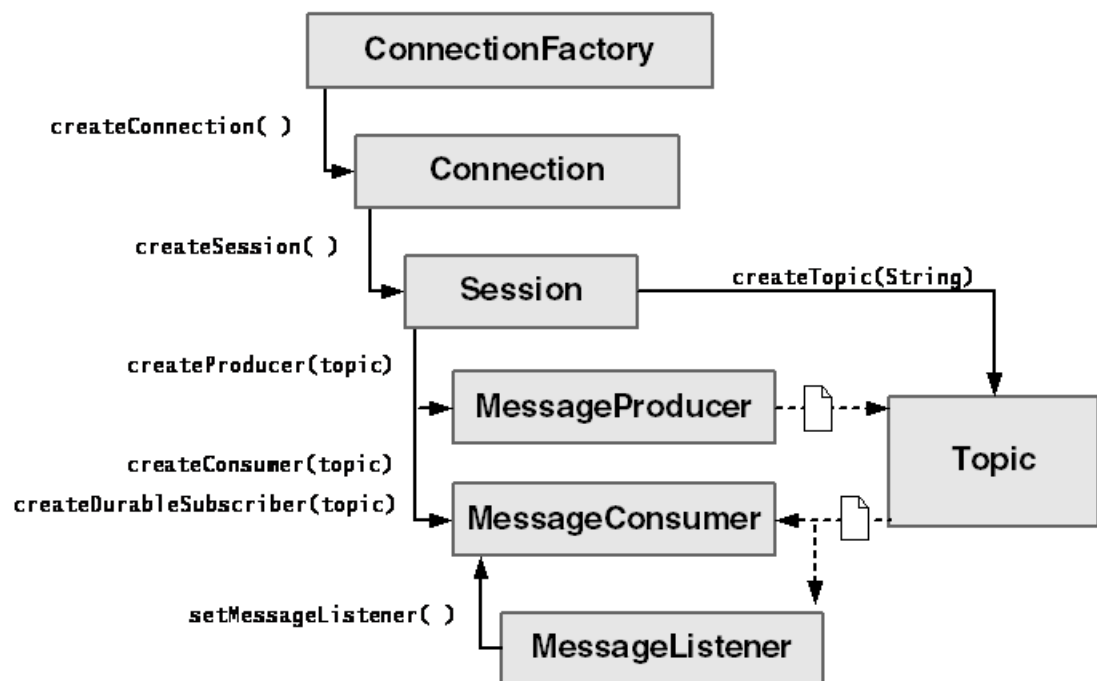
For details, see the following topics:

- [About Publish and Subscribe Messaging](#)
- [Message Ordering and Reliability in Pub/Sub](#)
- [Topic](#)
- [MessageProducer \(Publisher\)](#)
- [MessageConsumer \(Subscriber\)](#)
- [Durable Subscriptions](#)
- [Dynamic Routing with Pub/Sub Messaging](#)
- [Shared Subscriptions](#)
- [MultiTopics](#)

About Publish and Subscribe Messaging

The Publish and Subscribe (Pub/Sub) messaging model is shown in the following figure. In Pub/Sub messaging, a message is sent to a **topic**, and each consumer of that topic gets the message. This **one-to-many** model keeps topic producers (publishers) independent of the topic consumers (subscribers). In fact, producers could be sending messages to topics where no consumers exist.

Figure 58: 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 261.

See [Hierarchical Name Spaces](#) on page 345 for information about how SonicMQ applications can subscribe to sets of topics.

The following code snippet, from the **Chat** sample application, shows how to create the objects used in a **Session** for Pub/Sub communication: **Topic**, **MessageConsumer**, **MessageProducer**, and **Message**.

Creating Objects for Pub/Sub

```
//The topic is defined as a hierarchical topic
private static final String APP_TOPIC = "jms.samples.chat";

// Create Publisher and Subscriber to 'chat' topics
try{
    topic = pubSession.createTopic(APP_TOPIC);
    subscriber = subSession.createConsumer(topic, "SampleSubscription");
    subscriber.setMessageListener(this);
    publisher = pubSession.createProducer(topic);
    connection.start();
}catch (javax.jms.JMSEException jmse)
{
    jmse.printStackTrace();
}...
try
{
    // Read all standard input and send it as a message.
    java.io.BufferedReader stdin =
        new java.io.BufferedReader( new java.io.InputStreamReader( System.in ) );

    System.out.println("\nEnter text messages to clients that subscribe to the "
        + APP_TOPIC + " topic." +
        "\nPress Enter to publish each message.\n");

    while ( true )
    {
        String s = stdin.readLine();
        if ( s == null )
            exit();
        else if ( s.length() > 0 )
        {
            javax.jms.TextMessage msg = pubSession.createTextMessage();
            msg.setText( username + ": " + s );
            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](#) .

General Services

Asynchronous message delivery allows messages to be delivered with a range of options that ensure 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).

Reliable message delivery also deals with questions of ordering and redelivery.

Message Ordering

A predictable sequence of messages is a series of messages that have 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 Pub/Sub domains:

- 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 the 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 262.

Reliability

The assurance that a message will be received by a consumer has several other influences in Pub/Sub domains:

- 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 will receive. Regular subscriptions and durable subscriptions with a message selector definition that excludes a message will never get that message.

Message destruction due to expiration or administrator action (removing a durable subscription) permanently disposes of stored messages.

Topic

Topics are objects that provide the producer, broker, and consumer with a destination for JMS methods. 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 **java.lang.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, such as:

- . (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.
- | (vertical bar) is used only with MultiTopics.

See [Hierarchical Name Spaces](#) on page 345 for more information.

Important: [Connection Factories](#) on page 117 lists characters that are not allowed in SonicMQ names. Refer to this list for characters you can use in topic names.

You can programmatically store and retrieve topics in a directory service such as LDAP or the embedded Java Naming and Directory Interface (JNDI) service. With SonicMQ, you can store topic names in a JNDI or a simple file store, and then reference the object indirectly (by name) in some context. See [SonicMQ Connections](#) for more information.

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 need 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**:

```
publisher = session.createProducer(null);
topic = session.createTopic(jms.sample.chat");
publisher.send(topic, msg);
```

Creating the MessageProducer

This sample code creates a **MessageProducer** that specifies a default **Topic**:

```
javax.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 )
  {
    javax.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 **javax.jms.Message**
- **deliveryMode** is [NON_PERSISTENT|PERSISTENT|NON_PERSISTENT_SYNC|NON_PERSISTENT_ASYNC|NON_PERSISTENT_REPLICATED|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 that were 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** receives 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 **Session.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 the destination of the subscription.
- **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. If the durable subscriber is also part of a shared subscription, the message selector must match the selector of other members of the shared subscription.
- **noLocal** is a boolean. When set to **true**, the subscriber does not receive messages from subscribed topics that were published locally.

Note: SonicMQ extends the PubSub Message Consumer to enable shared subscriptions. When a **MessageConsumer** is also a member of a shared subscription, the **MessageConsumer** does not receive all messages published to the topic; instead, the **MessageConsumer** receives a subset of the messages, because delivery of the messages is load-balanced to all members of the shared subscription. See [Shared Subscriptions](#) on page 266 for more information.

The **TopicSubscriber** interface extends the **MessageConsumer** interface. Since **TopicSubscriber** is a JMS Version 1.02b model-specific interface, you should avoid using the interface directly, because the JMS Version 1.1 specification states that some of the model-specific interfaces might be deprecated in future versions. 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 combined with the user name and the client identifier to define the durable interest. This construct lets you create 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 [Connection Factories](#) on page 117 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** will throw a **JMSEException**.

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 that has been 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.

SonicMQ creates a common message store for all durable members of a shared subscription. If all members (durable and non-durable) are inactive, SonicMQ stores messages in the common store, until one or more members (durable or non-durable) becomes active. SonicMQ retains the common store until the last durable member unsubscribes, at which time the store is deleted.

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 will occur 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 has 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 JMS 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 JMS publisher sessions.

In a single broker configuration that does not use shared subscriptions, message ordering to durable subscribers is always guaranteed. In a clustered environment, SonicMQ supports optional strict message ordering to durable subscribers (unless they are members of a shared subscription). 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 get trapped on a crashed or partitioned broker. Applications that elect strict message ordering for durables, 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 will be 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 made at the topic session always takes precedence over any settings made 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:

```
progress.message.jclient.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:

```
progress.message.jclient.Session.setDurableMessageOrder (boolean
durableSubscriberMessageOrder)
```

It is possible to change the message ordering of a durable subscriber each time it connects. However, once the durable has connected 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

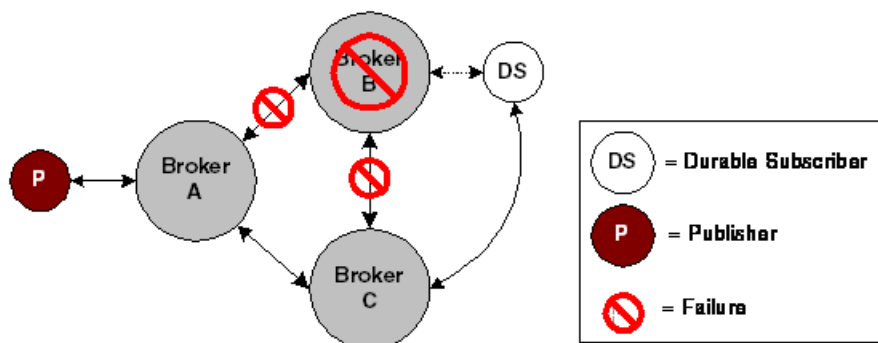
This section explains the availability of clusterwide durable subscriptions after a broker becomes unavailable and then reconnects. If a broker goes down as a result of a software or hardware failure, all messages that were stored on behalf of durable subscriptions on that broker become unavailable until 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 subscription's messages until that broker is restarted.

A client application will be able to publish messages as long it is connected to some broker in the cluster even if other brokers in the cluster are down.

The following figure 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 59: Clusterwide Durable Subscription Availability After Failure



Durable Message Ordering Enabled

In the above figure, broker **B** is unavailable and the 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** will be 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 that were 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 on **A** 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 has become unavailable.

Durable Message Ordering Disabled

When strict durable message ordering is not enabled in the example shown in the following figure, message delivery is not delayed for the durable subscriber **DS**. Messages stored on broker **B** will not be 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** will be sent to **DS** on **C**. Once the brokers reconnect to **B**, the skipped messages will be 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 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 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 results are a static list of connections instead of a static connection.

The 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. For more information on these topics, see the *Aurea 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 chapters “Configuring Routings” and “Managing SonicMQ Broker Activities” in the *Aurea SonicMQ Configuration and Management Guide* for information about how to perform these administrative tasks.

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 only when you want to deliver the message only to the subscribers of a single remote node. If your application messages are supposed to be delivered according to the global subscription rules that have been set up administratively, you should use the syntax **topic_name** as you normally would.

It is also possible to administratively connect the topic spaces of two nodes so that messages published to a topic on one node will be 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 need to be aware that they are sending to, or receiving from, topics that cross routing nodes.

The *Aurea 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 chapter “Multiple Nodes and Dynamic Routing” in the *Aurea 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 chapter “HTTP Direct Acceptors and Routings” in the *Aurea 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 was referenced when the application created the destination. For example, the destination name can be referenced in the following ways:

- **destination** (non-remote)
- **routing_node_name::destination** (topic on **routing_node_name**)
- **::destination** (topic on the current node)

Shared Subscriptions

A problem in JMS 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 farther and farther behind.

Three typical JMS solutions are:

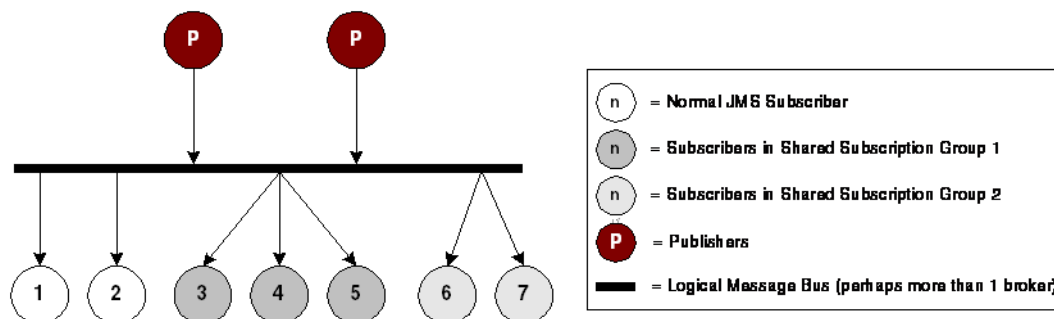
- **ServerSessionPools** — A single application server consumes messages on a single subscription in parallel. The shortcoming of this architecture is that all subscribers share one subscription in one JVM (for example, a J2EE AppServer) on one machine, which might be the bottleneck.
- **Forward messages to Queue** — A single application consumes messages from the topic and forwards the messages to a queue. JMS provides a model for shared **MessageConsumers** (reading from a queue) running in parallel on multiple machines. 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 gets every message and application logic must either serialize requests against a common resource such as a persistent storage mechanism, or check with central controlling program to resolve the duplicates.

SonicMQ provides a solution to the bottleneck problem by letting you establish groups of topic subscribers that share subscriptions to allocate the message load between them such 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 JMS 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.

The following figure 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 60: Illustration of Subscribers Abstracted from Specific Broker Connections



The following table shows how messages are received in the shared subscription configuration shown in the following figure. In the scenario shown in the following table, ten sequential messages are sent. The **X**'s in the table indicate which subscribers received and acknowledged each message. In this scenario, **Subscriber 6** (a non-durable subscription) does not acknowledge receipt of **Message 6** before it fails, so all subsequent messages for **Group 2** are delivered to the remaining member of that shared subscription group, **Subscriber 7**.

Table 28: Example of Messages Received Under Load Balancing (Continued)

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

If **Subscriber 6** were a durable subscription, **Message 7** and **Message 9** would have been stored for **Subscriber 6** until it reconnected to its durable subscription.

Storage of Messages for Durable Members of a Shared Subscription

SonicMQ creates a common message store for all durable members of a shared subscription. If all members (durable and non-durable) are inactive, SonicMQ stores messages in the common store, until one or more members (durable or non-durable) becomes active. SonicMQ retains the common store until the last durable member unsubscribes, at which time the store is deleted.

Features of Using Shared Subscriptions in Your Applications

Implementing shared subscriptions with groups of topic subscribers in your applications provides you with the following features:

- 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 JMS API methods.
- When any member of a group is connected, that member will receive new messages.
- Messages are allocated evenly to members of a shared subscription group. However, clients that are slow to the point of becoming flow-controlled are explicitly skipped in message allocation. In addition, delivery to local subscribers (connected to 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 though the group.
 - There is no guarantee that some members might not be allocated messages more than once in some cycles through the group.
 - Fairness is determined as a long-term average, rather than a short-term strict round-robin. The following table shows fair delivery for **Group 1** shown in the following figure.

Table 29: Balanced and Fair Delivery to a Shared Subscription Group

	Shared Subscription Group		
Message #	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

	Shared Subscription Group		
Message #	Subscriber 3	Subscriber 4	Subscriber 5
11	X		
12			X

Usage Scenarios for Shared Subscriptions

The following sections describe cases where implementing shared subscriptions can improve the performance of your applications.

Fault Resilience

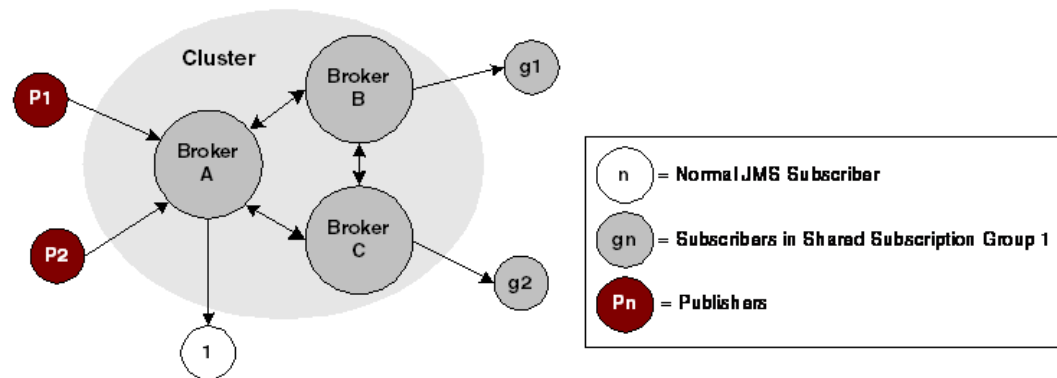
In the following figure, the goal is to guarantee that a topic subscriber application receives messages exactly once, and is resilient to both broker and application failures.

In this example, subscribers **g1** and **g2** are in a shared subscription group. The messages stream from publishers **P1** and **P2** are divided between the two of 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 JMS subscriber (**1**) receives all messages.

Figure 61: Fault Resistance Across Shared Subscription Topic Subscribers on a Cluster



Note: This configuration is **faultresistant** 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). Newer messages will be redirected entirely to topic subscriber **g2**.

Highly-Variable Processing Times

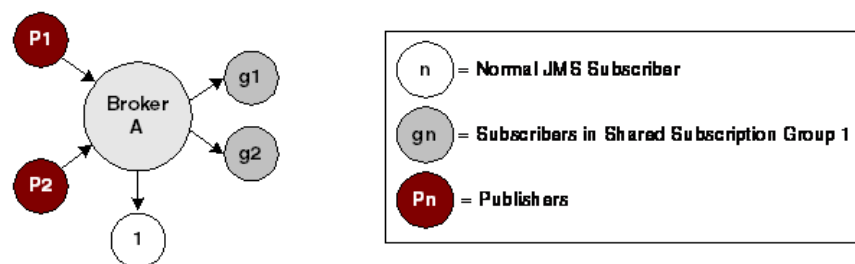
There are situations where a topic subscriber is fast enough to handle the message flow, but where some individual messages take significant processing. In a normal JMS application, the processing of messages following such a message must wait.

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 the following figure, 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 may 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** can not 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 62: Application in a Shared Subscription Group Processes Messages Once at Most



Pure Load-balancing

The throughput in Pub/Sub messaging is effectively limited to a speed of the slowest topic subscriber. If you want 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. If message selectors are used in the subscriptions, all the subscribers in the group must use the same message selector definition so that the end result of the shared effort is consistent and predictable.

Defining Shared Subscription Topic Subscribers

Use the following syntax to name topic subscribers within a group:

```
[[prefixName]]topicName
```

The validity of group names is done only on pure JMS clients and throws an **JMSEException** if the group name is invalid.

Once a topic has been created, a subscription can be created on that topic using normal JMS semantics, with the following additional requirements:

- The following call creates a topic object:

```
javax.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)
 - **|** (vertical bar)
- 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 nlocal, String selector);
```

 - A group is created when the group name and the topic name are identical.
 - For examples:
 - [[group1]]topic1** and **[[group1]]topic3** are distinct shared subscriptions.
 - [[group1]]topic1** and **[[group2]]topic1** are distinct shared subscriptions.
 - SonicMQ treats a shared subscription to a MultiTopic as separate from a shared subscription to an ordinary topic, even if the group name and topic name are the same. See [MultiTopics](#) on page 280 for more information
 - For example:
 - [[group1]]T1** and **[[group1]]MULTITOPIC:T1** are distinct shared subscriptions.
 - [[group1]]MULTITOPIC:T1||T2** and **[[group1]]MULTITOPIC:T1||T2||T3** are the same shared subscription.

- Similar to **QueueReceivers**, the **nolocal** parameter is ignored for shared subscription 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: Selector stings must match. Choose to use broker-side selectors.

- The following methods creates a durable subscriber on topic **T**, as part of the group:

```
javax.jms.Session.createDurableSubscriber (Topic T, String subscriptionName);  
  
javax.jms.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** would be delivered to one member of each of the following groups:
 - **[[group]]T.A**
 - **[[grp2]]T.A**
 - **[[group]]T.***

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 must match for all members of a shared subscription.
- Durable subscribers in the same group must follow normal JMS 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
- Similar to **QueueReceivers**, the **nolocal** parameter is ignored for shared subscription subscribers.
- 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:

```
javax.jms.Message.setJMSReplyTo(Topic T)
```

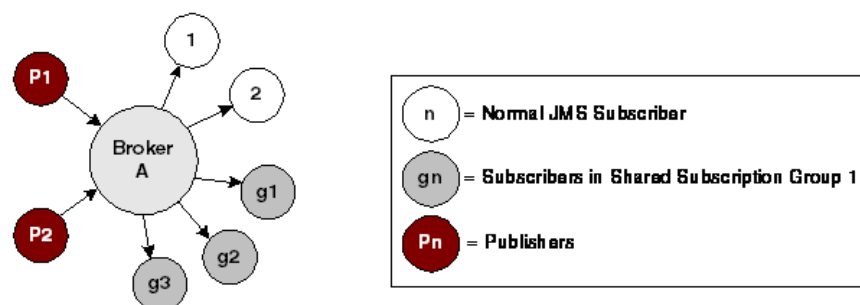
Message Delivery to a Broker with Shared Subscriptions

This section describes message delivery in both a single broker and cluster configurations.

Single Broker Behavior with Shared Subscriptions

In the following figure, 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 63: Shared Subscriptions with a Single Broker



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 gets the next published message, not the first un-processed message. Unprocessed messages are not reallocated. Once a message has been sent to a particular client context, it won't be reallocated unless the client fails.

Similarly, adding a new group member breaks a flow control situation. Existing clients that are flow controlled will 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 must use the same selector.

Disconnecting Group Member (Non-durable)

When a non-durable member of a shared subscription group is closed, or disconnects, all messages that have been allocated to the client might be discarded. 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 that have been allocated or delivered to the client. In addition, pending messages that have been allocated to a particular client are lost, even if they have not been delivered to the message listener's `onMessage()` method.

Shared Durable Subscriptions

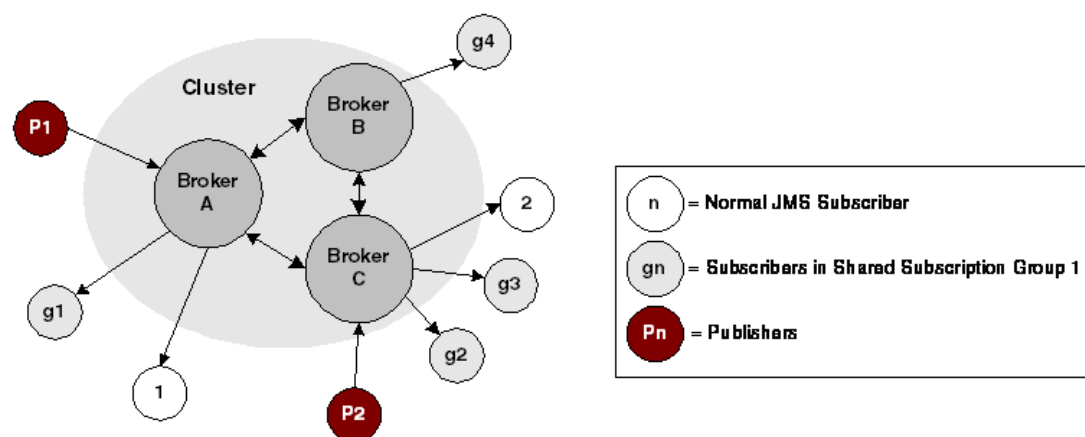
A shared subscription group uses a common message store for all members of the group. If there are durable members in the group and no group members are active, SonicMQ persists messages to the common store until at least one member (durable or non-durable) becomes active. When a member becomes active, SonicMQ delivers messages from the common store to the active member. As other members become active, SonicMQ load-balances message delivery to active members. SonicMQ retains the common store until the last durable member unsubscribes, at which time it deletes the store.

If a message is load-balanced to a durable member of a shared subscription, and the durable member closes without acknowledging the message, SonicMQ reallocates the message to another active member. This behavior means that the order of messages cannot be guaranteed for shared subscriptions that have durable members.

Cluster Behavior with Shared Subscriptions

SonicMQ supports shared subscriptions in a cluster. In the example shown in the following figure, 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 64: 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 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 message store I/O, it 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, the message is written to the group's message store.
5. If the broker cannot find any disconnected durable subscribers, it gives the message to the next available subscriber, even if it causes a flow-control situation to occur.

Once the broker allocates a message to a particular group member, the subscriber is expected to consume the message and acknowledge receipt. Otherwise:

- If the subscriber is non-durable, and it closes, unacknowledged messages are lost.
- If the subscriber is durable and it closes, unacknowledged messages are re-allocated to active (durable or nondurable) group members; if no other group member is active, the messages are saved to the group's message store.
- If the subscriber does a **rollback()** on a transacted session, or if it does a **Session.recover()**, messages are redelivered to that same subscriber.

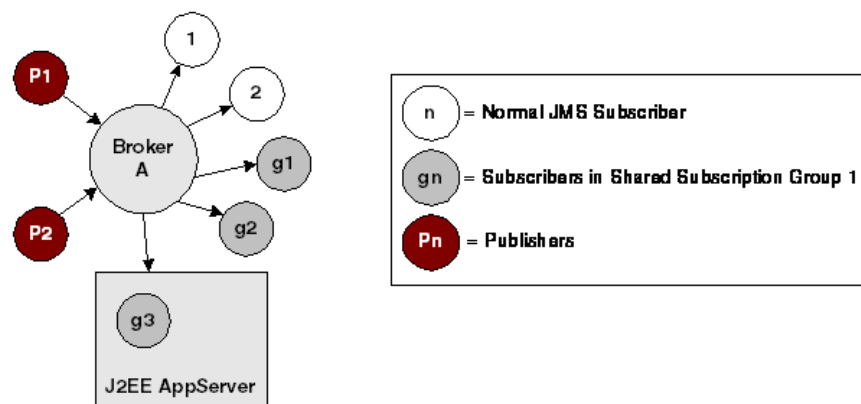
JMS Interactions with Shared Subscriptions

The following sections describe some examples of JMS interactions that can occur with shared subscriptions.

Connection Consumers

The example shown in the following figure involves multiple group members, some of which are connection consumers. In SonicMQ, a non-durable topic connection consumer allows a J2EE AppServer to have multiple threads on the client side processing messages. The SonicMQ broker sees this configuration as a single client context shipping messages to a single socket.

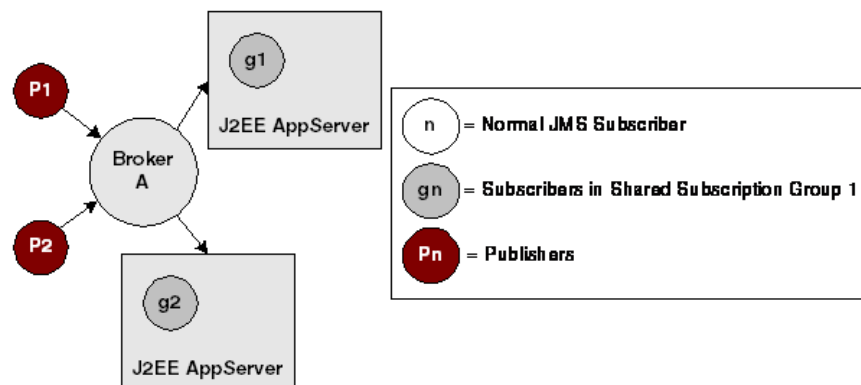
Figure 65: Connection Consumer in an AppServer



While the SonicMQ broker is able to determine that shared subscription group **g3** is a connection consumer, the broker does not determine how much faster that subscriber is than normal subscribers. If, for example, there are six threads handling messages at **g3**, a simple allocation of messages to each context will only give one third of the messages to that group. You can use connection consumers for shared subscriptions, but there is an assumption that all the subscriptions are similar in capabilities and configuration.

In the following figure, similarly configured connection consumers have comparable session pool sizes.

Figure 66: Similarly Configured Connection Consumers



Session Recovery

When the **recover()** method is called on a session, the normal JMS behavior occurs. That is, message delivery is stopped in the session, then restarted with the oldest unacknowledged message for that session. The redelivered messages will have their **JMSRedelivered** flag set to **true**—a setting that can be made only by the broker.

No reallocation of messages will occur from one shared subscription topic subscriber to another.

Transacted Sessions

A transacted session will delay acknowledgement of messages received on it until the **commit()** method is called. For non-durable subscriptions, closing or failing the session will cause those unacknowledged messages to be discarded.

For durable subscriptions, closing or crashing the session will cause 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, then 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 give no additional protection from message loss for these non-durable shared subscription topic subscribers. No reallocation of messages will occur from one load-balanced subscriber to another (except when the subscription is durable).

Shared Subscriptions with Remote Publishing and Subscribing

SonicMQ extends the dynamic routing architecture (DRA) to topics by allowing:

- **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 should go normally to normal subscribers, and round-robin to shared subscribers.

See [Dynamic Routing with Pub/Sub Messaging](#) on page 264 and the chapter “Multiple Nodes and Dynamic Routing” in the *Aurea SonicMQ Deployment Guide* for information about remote publishing and global subscriptions.

For global subscriptions, there are two points to note:

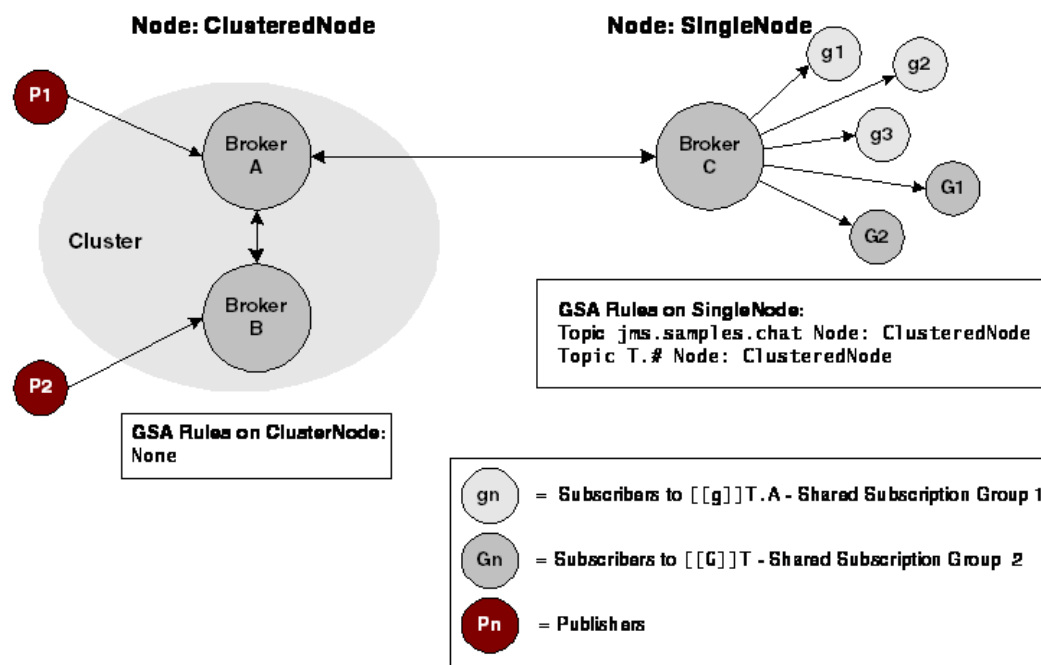
- Remote subscriptions are only valid for topics without the group prefix. That is, you can have a rule that propagates **T.A**, but not **[[g]]T.A**.
- The subscribing node may have multiple group subscribers, but these act like a single subscriber when triggering a rule. That is, if two subscribers 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.

The following figure shows an example that 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 67: 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 `json.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 has been defined that allows a subscription sent to a broker on one node, **SingleNode**, to be propagated to another node, **ClusteredNode** (see Figure 68). 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 68). 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 were no global subscriptions in the configuration shown in Figure 68, the messages sent by publishers on **ClusteredNode** would be 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 68:

- 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 68. 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 68). 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**.

MultiTopics

A MultiTopic combines multiple underlying component topics into a single destination. A message producer can use a MultiTopic to publish a message to multiple topics in a single operation (which is significantly faster than publishing to multiple topics individually). A message consumer can also use a MultiTopic to subscribe to multiple topics in a single subscription. (Alternatively, a message consumer can accomplish this via hierarchical namespaces and template characters. However, MultiTopics differ from hierarchical namespaces in how they define a set of topics for subscription. See [Hierarchical Name Spaces](#) on page 345)

MultiTopic publishing and MultiTopic subscribing are separate features. A message producer can publish to a MultiTopic even if no message consumer subscribes to the MultiTopic (although different message consumers can subscribe to different component topics within the MultiTopic). Similarly, a message consumer can subscribe to a MultiTopic even if no message producer publishes to it (although different message producers might publish to different component topics within the MultiTopic).

When a message producer publishes a message to a MultiTopic, the message is received by any message consumer subscribed to any of the underlying component topics. When a message consumer subscribes to a MultiTopic, the consumer receives messages published to any of its component topics.

Format of a MultiTopic String

A MultiTopic, like a standard topic, has a string representation. The format of this string is described below. MultiTopic strings are designed strictly for read-only purposes. Do not try to parse and manipulate MultiTopic strings.

MultiTopic String Format

The **javax.jms.Topic** interface defines a **getTopicName()** method, which returns a string. This interface is extended by the **progress.message.jclient.MultiTopic** interface, which overrides the **getTopicName()** method, returning a string with the following format:

```
MULTITOPIC:topicName_1[|topicName_2|topicName_N...]
```

where each **topicName** (**topicName_1**, **topicName_2**, and **topicName_N**) is a standard topic name.

Examples of MultiTopic Strings

The following examples are all valid MultiTopic strings:

- **MULTITOPIC:TopicA||TopicB||TopicC** — A MultiTopic with component topics **TopicA**, **TopicB**, and **TopicC**.
- **MULTITOPIC:TopicC||TopicA||TopicB** — A MultiTopic whose component topics are the same as the previous example but in a different order. This MultiTopic is functionally equivalent to the previous example; the ordering of component topics is immaterial.
- **NODEA::MULTITOPIC:TopicA||TopicB||TopicC** — A MultiTopic routed to **NODEA**. This MultiTopic can be used only for publishing (not subscribing).
- **[[group1]]MULTITOPIC:TopicA||TopicB||TopicC** — A MultiTopic where each component topic is part of a shared subscription (**group1**). This MultiTopic can be used only for subscribing (not publishing).
- **MULTITOPIC:TopicA** — A MultiTopic that results in receipt of the same messages that a subscription to **TopicA** would receive.

Note: See [Shared Subscriptions](#) on page 266 for more information on the intersection of the syntaxes for shared subscriptions and multitopics.

Creating MultiTopics

When you create a **MultiTopic** object, it is initially empty and has no component topics. There are two ways to create a **MultiTopic** object, using a **Session** object or a **DestinationFactory** object.

Using a Session Object to Create a MultiTopic

You can use a **progress.message.jclient.Session** object to create an empty **MultiTopic** object. The following code snippet, from the **MultiTopicChat** sample, shows how to do this:

```
//Create an empty MultiTopic from the subscriber session.
progress.message.jclient.MultiTopic multiTopic =
((progress.message.jclient.Session) subSession).createMultiTopic();
```

Using a DestinationFactory Object to Create a MultiTopic

If you are writing an application that needs to create a MultiTopic but does not have access to a **Session** object, you can use a **progress.message.jclient.DestinationFactory** to create the MultiTopic. The **DestinationFactory** interface defines a **createMultiTopic()** method that creates an empty **MultiTopic** object. The following code snippet, from the **MultiTopicChat** sample, shows how this is done:

```
//Create an empty multitopic using the SonicMQ DestinationFactory:
progress.message.jclient.MultiTopic ret =
    progress.message.jclient.DestinationFactory.createMultiTopic();
```

Adding Component Topics to a MultiTopic

After you create an empty **MultiTopic** object (using either a **Session** or **DestinationFactory**), you can use the **MultiTopic.add(Destination dest)** method to add component topics. When you call this method, the type of **Destination** objects have the following constraints:

- You cannot add a Queue.
- You cannot add a topic with a different node name than existing topics.
- You cannot add a topic with a different shared subscription name than an existing topic.
- You cannot add both a node prefix (used for publishing) and a shared subscription name (used for subscribing) to the definition of a destination.

You can add a MultiTopic to a MultiTopic; this is equivalent to adding each component topic individually.

You can add a TemporaryTopic to a MultiTopic. However, you cannot create a durable subscriber on a MultiTopic that contains one or more temporary topics. If a temporary topic is deleted, a subsequent attempt to publish using a MultiTopic that contains the deleted temporary topic fails immediately with an **InvalidDestinationException**. When a MultiTopic is saved to an object store, its component temporary topics are not serialized.

Publishing and Subscribing to MultiTopics

You publish to a MultiTopic in much the same way as you publish to an ordinary topic; and you can subscribe to a MultiTopic in much the same way as you subscribe to an ordinary topic. However, there are several issues to consider regarding MultiTopic publishing and subscribing, described in the following sections.

Splitting MultiTopic Delivery

With MultiTopic publishing, a message producer can publish a message to multiple component topics, which means that a single message consumer can receive the same message multiple times. This can occur if the message consumer subscribes to all or some of the MultiTopic's component topics, either by subscribing to another MultiTopic (whose component topics overlap those of the message producer's MultiTopic) or by subscribing to a set of topics in a hierarchical namespace (whose topics overlap those of the message producer's MultiTopic).

For example, a message producer can publish a message to the following MultiTopic:

```
MULTITOPIC:foo.a||foo.b||foo.c
```

In this situation, there are two different ways that a message consumer can receive multiple copies of the message: the message consumer can subscribe to a MultiTopic with overlapping component topics (for example, **MULTITOPIC:foo.a||foo.c**); or the message consumer can subscribe to a set of topics in a hierarchical namespace (for example, **foo.***).

A message consumer in this situation can control whether to receive multiple copies or a single copy of the message. The message consumer controls this by calling the **setSplitMultiTopicDelivery(boolean value)** method on either a **Session** or on a **ConnectionFactory**. When called on a **Session**, it affects only that session. When called on a **ConnectionFactory**, it affects all sessions created from the **ConnectionFactory**.

By default, the value of **SplitMultiTopicDelivery** is **false**. A MultiTopic subscriber receives only one copy of a message from a MultiTopic publish, even when it is deliverable through several topics that match topics in the publisher's list. In this message, the value of the **JMSDestination** header is a MultiTopic. This MultiTopic includes any component topics overlapping those of the message producer's MultiTopic. For example, if the message producer publishes to **MULTITOPIC:foo.a||foo.b||foo.c**, and the message consumer subscribes to **foo.***, the value of the **JMSDestination** header is **MULTITOPIC:foo.a||foo.b||foo.c**. If the message consumer subscribes to **MULTITOPIC:foo.a||foo.c**, the value of the **JMSDestination** header is **MULTITOPIC:foo.a||foo.c**.

When the split delivery **value** is set to **true**, new subscribers take delivery of one copy of a message MultiTopic published for each topic in its MultiTopic list that match topics in the publisher's list. In each message, the value of the **JMSDestination** header is the component topic where the message was published.

Important: Set the split delivery option before creating subscribers. Once subscribers are created, changes to this setting are not applied.

Remote Publishing

A message producer can publish a message destined for multiple topics on a remote node via one method call. However, a message producer cannot publish a message to more than one node using a single MultiTopic.

If a message producer publishes to a remote node using a MultiTopic, and an ACL check on the remote node fails for a component topic, the message is not published to that component topic. However, the message is published to any component topic that passes an ACL check. For example, if the message producer on NodeA publishes a message to

NodeB::MULTITOPIC:topic1||topic2||topic3, and ACL check fails on **topic2**, the message is published only to **topic1** and **topic3**.

For more information about remote publishing, see the “Multiple Nodes and Dynamic Routing” chapter in the *Aurea SonicMQ Deployment Guide*.

Global Subscriptions

A subscription created in a publishing node on behalf of a subscribing node is referred to as a global subscription. The subscribing node requests creation of a global subscription when a local subscriber connects to the subscribing node.

For more information about global subscriptions, see the “Multiple Nodes and Dynamic Routing” chapter in the *Aurea SonicMQ Deployment Guide*.

Global subscription rules on a subscribing node can specify topics for which global subscriptions on the publishing node are created. These global subscriptions allow the subscribing node to effectively subscribe to the publishing node and deliver to its own subscribers the messages it receives from the publishing node.

Important: A global subscription rule cannot use a MultiTopic in its definition.

When a message consumer subscribes to a MultiTopic on a subscribing node, the subscribing node checks each of the MultiTopic's component topics against the subscribing node's global subscription rules. If an appropriate global subscription rule is in place, a global subscription is created on the publishing node and the subscribing node receives messages from the publishing node for that destination. However, the subscribing node receives messages only for topics that pass ACL checks on the publishing node. If some topics fail ACL checks at the publishing node, it does not result in the failure of all topics.

MultiTopics and Access Control Lists (ACLs)

You cannot define ACLs for a MultiTopic; you define ACLs for the underlying component topics.

If a message producer publishes a message to a MultiTopic on its local node, the publish operation fails entirely if an ACL check fails for any component topic. The message is not published to component topics that pass the ACL check.

If a message consumer subscribes to a MultiTopic on its local node, the subscription fails entirely if an ACL check fails for any component topic. Messages are not received on component topics that pass the ACL check.

If an ACL is changed that denies subscribe permission on a component topic for an existing MultiTopic subscription, the subscriber no longer receives messages on that topic.

MultiTopic Considerations

When you use MultiTopics, there are several issues to consider, described in the following sections.

JMSReplyTo

You cannot specify a MultiTopic as the destination value for **JMSReplyTo**. An attempt to do so will throw a **JMSEException**.

QoP and Per Message Encryption

The Quality of Protection (QoP) of a message is determined by choosing the highest level of protection for any component destination. If per message encryption is enabled for the message, it overrides any broker-defined value.

You cannot define QoP on a MultiTopic. QoP is defined for each component topic.

QoP Cache Size

A cache on each client connection to a broker buffers the QoP setting communicated from the broker when a message has been sent to a destination. When you use MultiTopics, a large list might keep this cache in a constant state of renewal for QoP settings.

You can modify the cache size after you create a connection to better handle the expected number of topics. See [Setting QoP Cache Size](#) for information on setting this parameter.

Durable Subscriptions

Changing the split delivery value on a subsequent reconnect will not result in the subscription being unsubscribed.

A DurableSubscriber that is subscribed on a MultiTopic can reconnect with a different set of topics as long as the new MultiTopic has at least one topic in common with the previous subscription. In this case messages stored on behalf will not be deleted. Instead they will be filtered as they are delivered to remove any Topics that no longer match the subscription. Note that the criteria for having one topic in common is that the topics exactly match. For example a change from **MultiTopic:Topic.1||Topic.2** to **MultiTopic:topic.*** would result in an unsubscribe while a change from **MultiTopic:topic.1||topic.2** to **MultiTopic:topic.1||topic.3** would not.

Switching a DurableSubscriber from a MultiTopic to a single topic subscription will result in the durable being unsubscribed. Correspondingly, switching a DurableSubscriber from a single topic subscription to a multitopic subscription will result in the durable being unsubscribed.

Shared Subscriptions

A message consumer can be a member of a shared subscription to a MultiTopic, provided all component topics are part of the same shared subscription.

The following MultiTopic string indicates a shared subscription:

```
[[group1]]MULTITOPIC:topic1||topic2
```

When a subscriber defines a single topic as a MultiTopic destination, a subscriber that does not use MultiTopic syntax would receive the same messages. However, the two syntaxes cannot define the same shared subscription. For example, **[[group1]]topic1** and **[[group1]]MULTITOPIC:topic1** are not equivalent, and therefore not a shared subscription for the two subscribers.

Note: It is not valid to imbed a **[[group]]** prefix in a MultiTopic definition. The following is not valid a string for a shared multitopic subscription: `[[group1]]MULTITOPIC: [[group1]]topic1`

A MultiTopic subscription group is defined by the set of intersecting topics for any group prefix. For example, consider two message consumers, each subscribed to one of the following MultiTopics:

```
[[group1]]MULTITOPIC:topic1||topic2
[[group1]]MULTITOPIC:topic1||topic3
```

Because the group prefix is the same, both subscriptions are considered part of the same shared subscription, but only on their intersecting topic **topic1**. Messages that would be delivered to **topic2** and **topic3** subscribers are discarded. When a message consumer subscribes with a smaller set of topics, the shared subscription is reduced.

Note: It is a good practice for all group members to use the same MultiTopic, perhaps maintained and accessed as an administered Destination object.

Non multi-topic group subscriptions are load balanced separately from MultiTopic subscriptions.

Note: See the chapter “Managing SonicMQ Broker Activities” in the *Aurea SonicMQ Configuration and Management Guide* for information about viewing disconnected shared durable subscriptions to multitopics.

HTTP Direct

You can specify a MultiTopic for HTTP Direct inbound. The MultiTopic format is recognized and handled for inbound JMS over HTTP requests. If reply to is specified as a MultiTopic an errorcode is returned.

Basic and SOAP

One way HTTP **PUTS** support the ability to configure the destination as a MultiTopic.

Content-reply and outbound HTTP do not allow MultiTopic destinations.

Flow Control

Multi-publishes are subject to normal pub/sub flow control. If any prior publish causes the publisher to flow control, a publish operation may block until there is room on the publisher's output queue. It will be up to the Administrator to determine which slow subscribers are problematic by use of the pubpause notification. If the subscriber has a Multi-subscription all subjects from that subscription will be listed in the notification.

Since messages that are routed to other brokers are placed in the routing queue, it is possible that the routing queue will fill up. If this happens the application will be flow controlled on all component topics until space becomes available on the routing queue.

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 will not be discarded until a client has processed them.

For details, see the following topics:

- [Introduction](#)
- [Duplicate Message Detection Overview](#)
- [Dead Message Queue Overview](#)
- [Handling Undelivered Messages](#)
- [Specifying a Destination for Undelivered Messages](#)
- [Undelivered Message Reason Codes](#)

Introduction

SonicMQ can guarantee message delivery when the broker system to which a client connects can be certain 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 has been 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 at the JMS client or application level:

- If there is a connection or network failure between the JMS 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 would never know that the message had been sent.
- The application might fail after the commit has occurred. Even though the message has been sent and committed at the SonicMQ level, the application would have 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 extends the JMS concept of a transacted session to allow 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 the following table.

Table 30: Session.commit Behavior

Condition	Action
transactionID exists	Throws an exception.
transactionID does not exist	Store a new value of transactionID and continue with normal Session.commit() behavior.

Note: The same duplicate detection **transactionID** table is used for JMS-for-HTTP (with HTTP Direct) and for large message support.

Support for Detecting Duplicate Messages

The SonicMQ message broker stores the index for the duplicate detection in a persistent storage mechanism. The mechanism is always created when the storage is initialized. The mechanism name is created from the **BrokerName** by default, but you can also explicitly specify this name if you choose.

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 of **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 *Aurea SonicMQ Configuration and Management Guide* for information about setting broker properties in the Management Console.

Note that every indexed commit requires a persistent storage mechanism action, so sessions using duplicate message properties will be significantly slower than other sessions. These persistent storage mechanism actions are sequentially committed, so if two JMS clients use the same ID at the same time, only one will be successful. There is no window where the two clients will both succeed.

Dead Message Queue Overview

Messages that expire or are viewed by SonicMQ as undeliverable are called dead messages. One type of dead message you encounter are those that expire. Other types of dead messages can occur in multi-node deployments, which are discussed in the *Aurea SonicMQ Deployment Guide* in the chapter “Multiple Nodes and Dynamic Routing.”

SonicMQ provides the Dead Message Queue (DMQ) to help you handle dead messages. Both topic and queue messages can be sent to the DMQ. In PTP messaging, undeliverable and expired 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 applications 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 and the SonicMQ broker finds messages that have exceeded their time to live (TTL) and should expire, or that cannot be routed due to some external network error, the broker saves the message in a dead message queue (DMQ) and/or generates an administrative notification (management event)

At an application level, you can listen for the administrative notifications, browse the DMQ, and deal with undelivered messages as appropriate. 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 will be 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 will be 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 to **true**.

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](#), [Dynamic Routing with Pub/Sub Messaging](#), and the chapter “Multiple Nodes and Dynamic Routing” in the *Aurea SonicMQ Deployment Guide* for information about Aurea 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 have been forwarded to another routing node, but where the handshaking needed to ensure once-and-only-once delivery of messages has been 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.
- **Undelivered messages** — Queue messages that have exceeded their delivery limit.

There are other reasons why a message might not be delivered, including timeouts and network failures. See the chapter “Multiple Nodes and Dynamic Routing” in the *Aurea SonicMQ Deployment Guide* for descriptions of various scenarios under which messages are not delivered. See [Undelivered Message Reason Codes](#) on page 302 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

In SonicMQ if your application specifies the DMQ option for each message, then all expired or undeliverable messages are sent to the DMQ, named **SonicMQ.deadMessage**. The DMQ is treated exactly like a normal queue in that it can be browsed or read using normal JMS 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). Topic messages that have a **JMSDeliveryMode** of **DISCARDABLE** are not saved to the DMQ.

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 enqueueing 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 292 for information about the message properties that request enqueueing 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 will raise 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 *Aurea 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 **(javax.jms.Message) m.getJMSDestination()** for the original topic or queue.

Default DMQ Properties

By default, SonicMQ creates the DMQ with the properties listed in the following table.

Table 31: Dead Message Queue Properties (Continued)

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 *Aurea 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 *Aurea 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 noted as being undeliverable. (Ignored for **DISCARDABLE** topic messages.) See [Setting the Message Property to Preserve If Undelivered](#) on page 293.

Note: If a routing user does not have permissions to write to the DMQ, messages arriving from this routing node will be dropped regardless of their **JMS_SonicMQ_preserveUndelivered** property (the messages will 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 as undeliverable. The broker sets this property when messages are moved to a dead message queue.

These property names are available as standard constants in **progress.message.jclient.Constants**. The following table provides the values for these constants.

Table 32: JMS SonicMQ Properties (Continued)

JMS 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:

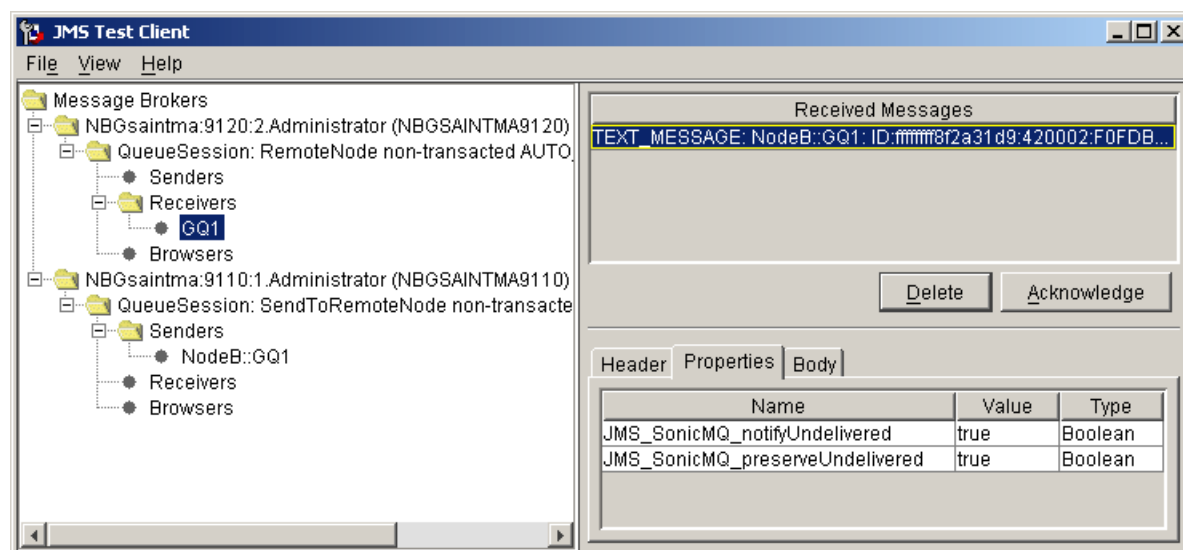
```
// Static setup
private static String Q_NAME = <Various>

// 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.
javax.jms.Queue theQueue = session.createQueue(Q_NAME);
javax.jms.MessageProducer sender = session.createProducer(null);
sender.send (theQueue, msg);
```

The following figure shows a message received on a global queue on the remote broker with the property that retains the message in the DMQ set to true. Notice that the message identifier at the top of the right panel indicates the global routing taken by this message.

Figure 68: Dynamically Routed Message That Requested to be Preserved



Handling Undelivered Messages

The following sequence of events describes the process SonicMQ uses to handle undeliverable messages:

1. A condition occurs where the broker determines the message is not deliverable.
(See Undelivered Message Reason Codes 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 **DISCARDABLE** 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 302 for a description of **reason_code**.

4. The special processing object determines whether to send a notification that the message has been sent to the DMQ or that the message is a queue message that has 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 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](#) on page 294
- [Using High Priority and Throwing an Admin Notice](#) on page 295

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. The following code snippet shows how this might be done.

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.
javax.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 final Strings in
// progress.messages.jclient.Constants.
msg.setBooleanProperty("JMS_SonicMQ_preserveUndelivered", true);
msg.setBooleanProperty("JMS_SonicMQ_notifyUndelivered", true);
// Send the message with PERSISTENT, TimeToLive values.
qsender.send(msg,
              javax.jms.DeliveryMode.PERSISTENT,
              javax.jms.Message.DEFAULT_PRIORITY,
              7200000);
```

Using High Priority and Throwing an Admin Notice

The following code snippet 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.

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.
javax.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 final Strings in
// progress.messages.jclient.Constants.
msg.setBooleanProperty("JMS_SonicMQ_notifyUndelivered", true);
// Send the message for fast delivery, or not at all.
qsender.send(msg,
              javax.jms.DeliveryMode.NON_PERSISTENT,
              8, // Expedite at a high priority
              600000); // 10 minutes
```

What To Do When the Dead Message Queue Fills Up

When the DMQ fills up (to its maximum queue size), the broker stops processing messages after enqueueing the message that caused the DMQ to exceed its maximum size. In this way, no messages are lost. See the “Configuring Queues” chapter in the *Aurea SonicMQ Configuration and Management Guide* for information about setting the maximum DMQ size and restarting the broker after a broker shutdown when the maximum DMQ size is reached.

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 final static int** in the **progress.message.jclient.Constants** class.

In this case, the SonicMQ broker determines that a message has expired. This failure type applies only to queue messages.

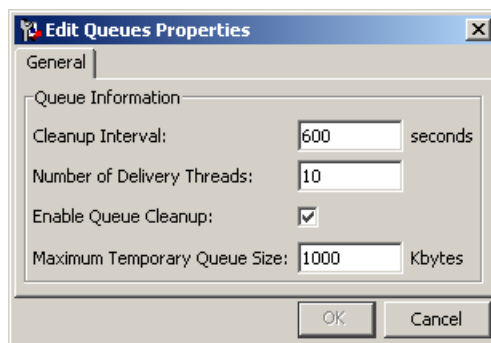
This dead message event is the simplest case and the one that most developers consider when thinking about dead message queues.

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 decide not to deliver the message due to expiration.

Checks for expiration are done only periodically within a broker (in order 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 Management Console in the **Edit Queues Properties** dialog box, as shown in the following figure. See the “Configuring Queues” chapter in the *Aurea SonicMQ Configuration and Management Guide* for information about using the Management Console to set the queue cleanup parameters.

Figure 69: 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 chapter “Multiple Nodes and Dynamic Routing” in the *Aurea SonicMQ Deployment Guide* for information about these topics.

Specifying a Destination for Undelivered Messages

By default, SonicMQ brokers use their predefined, system queue **SonicMQ.deadMessage** to preserve undelivered messages. You can override this default behavior by specifying an alternative destination for undelivered messages. Any non-system queue or topic can be used for this purpose, including temporary topics and temporary queues.

If you specify an alternative undelivered destination, it is critical that your application monitor the undelivered destination and promptly process its messages. Doing so avoids filling up the in-memory areas used for the undelivered destination at the SonicMQ broker.

How to Specify an Undelivered Destination

The following code snippet shows how an application can override the DMQ and specify an alternative undelivered destination.

Overriding the DMQ

```
String undelQueueName = "MyUndeliveredQueue";
javax.jms.Session session = //get a session from a JMS connection.
javax.jms.Destination undeliveredQueue = session.createQueue(undelQueueName);
```



```
progress.message.jclient.Message message = session.createTextMessage() ;
String undelPropName = progress.message.jclient.Constants.DESTINATION_UNDELIVERED
;
message.setDestinationProperty(undelPropName, undeliveredQueue);
```

The previous example uses the **createQueue** method to create a **Queue** object dynamically. More commonly, applications use a **Queue** or **Topic** object created by an administrator as a JMS administered object and stored in a JNDI namespace.

Applications can also use **TemporaryTopic** and **TemporaryQueue** objects created by the corresponding methods in the **javax.jms.Session** interface.

SonicMQ validates the destination in the **setDestinationProperty()** method as follows:

- It can be set for either topic or queue messages.
- It can be a Queue or a Topic.
- It can be a global queue that resides in a different routing node or in the local node.
- It can be a remote topic such as **Node::Topic**.
- It can be a clustered queue, global or not global.
- It can be a temporary queue or a temporary topic.
- It can be the **SonicMQ.deadMessage** queue at the broker where the application is connected.

However, an undelivered destination:

- Cannot be an Outbound HTTP Direct destination.
- Cannot be **SonicMQ.deadMessage** with a node prefix where the node is not the local node. If a remote **SonicMQ.deadMessage** is specified, an exception is thrown. If the local node is used as a prefix, the **SonicMQ.deadMessage** queue at the local broker is used.
- Cannot be **SonicMQ.routingQueue**, with or without a node prefix. If **SonicMQ.routingQueue** is specified, an exception is thrown.
- Cannot be the same as the actual destination of the message. If it is the same, an exception is thrown during the send call.
- Cannot be a **MULTITOPIC** construct.

SonicMQ uses the undelivered destination only if the **JMS_SonicMQ_preserveUndelivered** message property is set to **true**; otherwise, SonicMQ does not use the undelivered destination.

JMS_SonicMQ_destinationUndelivered Message Property

The following code snippet shows how an application can use the constant **progress.message.jclient.Constants.DESTINATION_UNDELIVERED** to specify a message property when calling the **message.setDestinationProperty()** method. The message property specified by this constant is the **JMS_SonicMQ_destinationUndelivered** message property.

Applications can use the **Message.getStringProperty()** method to retrieve the value of this property. The returned value is a string formatted as follows:

- **\$Q. + undeliveredQueueName** (if the destination is a queue)
- **undeliveredTopicName** (if the destination is a topic)

For example, where the undelivered destination is a queue named **myUQ**, the syntax is:

```
message.setStringProperty("JMS_SonicMQ_destinationUndelivered", "$Q.myUQ");
```

Note: Clients using version 6.1 or (earlier versions) do not support the Undelivered Destination feature, even when setting the above message property with **setStringProperty()**.

An application can also use the **Message.setStringProperty()** method to set the value of the **JMS_SonicMQ_destinationUndelivered** property, but this is not recommended. The correct way to set this property is to use the **Message.setDestinationProperty()** method.

When applied to a property whose value is a destination, the **Message.getObjectProperty()** method returns the value of the property as a string, which has the format described above. Likewise, the **Message.setObjectProperty()** method also expects a string value for a property of the destination type (and the string must also have the format described above).

If an application needs to copy message properties from one message to another it can use the **Message.setObjectProperty()** method, as shown:

```
Enumeration properties = message1.getPropertyNames();
while (properties.hasMoreElements())
{ String propName = properties.nextElement();
  message2.setObjectProperty(message1.getObjectProperty(propName));
}
```

The **Message.setObjectProperty()** method should be used for destination properties only to copy a property from one message to another. In order to set a destination property in the original message applications should use the **Message.setDestinationProperty()** method.

The **Message.isDestinationProperty()** method enables your application to test whether a property string is a destination. This method returns **true** if the type of the property is **javax.jms.Destination**; otherwise, it returns **false**.

An application can use this method to loop through a list of message properties obtained by the **Message.getPropertyNames()** method. If a property is a destination property, the application can use the **Message.getDestinationProperty()** method to retrieve the value of the property. Otherwise, the application can use the **Message.getObjectProperty()** method.

The **Message.isDestinationProperty()** method returns **true** only if the name of the property is either **JMS_SonicMQ_destinationUndelivered** or **JMS_SonicMQ_undeliveredOriginalJMSDestination**.

Changes to JMS Headers

When SonicMQ cannot deliver a message to its original destination, SonicMQ delivers the message to the undelivered destination. However, before delivering the message to the undelivered destination, SonicMQ modifies the values of the **JMSDestination**, **JMSTimestamp**, and **JMSExpiration** message headers:

- **JMSDestination** — SonicMQ sets this header to the destination specified by the **JMS_SonicMQ_destinationUndelivered** property of the message.
- **JMSTimestamp** — SonicMQ sets this header to the time when SonicMQ determined that the message could not be delivered to its original destination and has to be forwarded to its undelivered destination.
- **JMSExpiration** — SonicMQ sets this header to **0** (which means that the message does not expire).

SonicMQ only changes these JMS headers if the message-producing application overrides the system DMQ using the **JMS_SonicMQ_destinationUndelivered** message property. Otherwise, SonicMQ places the undelivered messages in the system DMQ and leaves their JMS headers unchanged. These headers also remain unchanged if SonicMQ cannot deliver an undelivered message to the specified undelivered destination for the reasons described in [Undelivered Message Reason Codes](#) on page 302

In case an application that receives the message from the undelivered destination needs to know the original values of the modified headers, SonicMQ copies the original values into corresponding message properties (described in the following section).

Message Properties for Undelivered Destinations

SonicMQ uses several message properties to support undelivered destinations. If a message becomes undelivered and has to be preserved in a destination other than the DMQ, SonicMQ sets the following message properties in the message (in addition to the **JMS_SonicMQ_undeliveredReason** and **JMS_SonicMQ_undeliveredTimestamp** properties):

- **JMS_SonicMQ_undeliveredBrokerName** — (String) Name of the broker where the delivery failure took place.
- **JMS_SonicMQ_undeliveredNodeName** — (String) Name of the routing node where the delivery failure took place.
- **JMS_SonicMQ_undeliveredReasonAddedToDMQ** — (Integer) Reason the message was added to the DMQ instead of the specified undelivered destination (see [Failure to Forward Undelivered Messages to the Undelivered Destination](#) on page 300).

Your application can use these message properties to determine the reason the message couldn't be delivered to its destination. SonicMQ sets these message properties even if the message is preserved in the DMQ. This allows applications that listen on undelivered messages to be written uniformly.

If SonicMQ sends a message to the undelivered destination, SonicMQ sets the following message properties before delivering the message to the undelivered destination:

- **JMS_SonicMQ_undeliveredOriginalJMSDestination** — SonicMQ sets this to the original value of the **JMSDestination** header.
- **JMS_SonicMQ_undeliveredOriginalJMSTimestamp** — SonicMQ sets this to the original value of the **JMSTimestamp** header.
- **JMS_SonicMQ_undeliveredOriginalJMSExpiration** — SonicMQ sets this to the original value of the **JMSExpiration** header.

Undelivered Messages and Message Expiration

If a message has expiration set and it can't be delivered, the message won't expire after it is forwarded to the undelivered destination. Therefore, applications must monitor the undelivered destination and promptly process its messages. This avoids filling up the in-memory areas used for the destination at the SonicMQ broker.

Failure to Forward Undelivered Messages to the Undelivered Destination

SonicMQ tries to add an undelivered message to the specified undelivered destination. If it fails to do so, SonicMQ adds the message to be added to the system DMQ instead. SonicMQ adds an undelivered message to the system DMQ instead of the specified undelivered destination in the following situations:

- If the undelivered destination is a queue and that queue does not exist (reason code **UNDELIVERED_JMS_QUEUE_NOT_FOUND**).
- If the undelivered destination is a queue and that queue is full (reason code **UNDELIVERED_QUEUE_FULL**).
- If the undelivered destination is a queue and the size of the message exceeds the max size of that queue (reason code **UNDELIVERED_MESSAGE_TOO_LARGE_FOR_QUEUE**).
- If the undelivered destination is a topic and at least one of the subscribers for that topic is flow controlling its publishers (reason code **UNDELIVERED_TOPIC_FULL**).
- Note that even if a subscriber uses FlowToDisk, it may flow control its publishers (for example, if the maximum topic database size has been exceeded).
- If some of the subscribers are using globalk subscriptions connected at another routing node (this situation may take place if the routing queue is full).
- If the undelivered destination resides in a different routing node and the message can't be delivered there because of a DRA-related problem (reason code is one of the DRA-related codes in the **progress.message.jclient.Constants** class).
- If the undelivered destination is an HTTP Direct destination (reason code is **UNDELIVERED_UNSUPPORTED_OVERRIDE_DESTINATION**).
- If the undelivered destination wasn't set by a compatible client application using a correct setter method (reason code is **UNDELIVERED_INVALID_PROPERTY_TYPE**).

If SonicMQ places an undelivered message in the DMQ instead of the specified undelivered destination, SonicMQ sets the **JMS_SonicMQ_undeliveredReasonAddedToDMQ** message property to the corresponding reason code as described above.

Note that in some cases, SonicMQ can detect the failure as soon as a message becomes undelivered—in that case, SonicMQ preserves the message in the DMQ at the broker processing the undelivered message. However, if the specified undelivered destination requires SonicMQ to send the message to another broker or node, the failure may take place at a different broker and SonicMQ places the message in the DMQ at that broker.

Publish Permission Check

If an application that uses the undelivered destination property is connected to a secure broker, the broker verifies that the application's credentials have been granted the publish permission to the specified destination.

If the permission check fails, the message is rejected by the broker and the SonicMQ client runtime throws a **JMSSecurityException**.

The publish permission check on the undelivered destination is performed as soon as the broker receives a message - therefore, if the check fails, the application receives an exception even though the message has never become undelivered.

The publish permission check on the undelivered destination is performed whenever the undelivered destination property is set in the message received by the broker, even if the **JMS_SonicMQ_preserveUndelivered** property is not set or is set to **false**.

In the DRA environment, the publish permission check is done by every broker that receives the message as it is being routed to its destination. Since each node can use its own authentication and its own security policy, it is possible that the message is received successfully by the local broker but is later rejected by the remote node. In that case, the application doesn't receive an exception but the message is dropped at the remote node.

Whenever the publish permission check on the undelivered destination fails, the broker writes an error message in its log as it always does for the publish permission check on the message destination.

Undelivered Message Notifications

If a message that is declared to be undelivered by SonicMQ, has the **JMS_SonicMQ_notifyUndelivered** message property set to **true** and the **JMS_SonicMQ_destinationUndelivered** property of the message is not null, the specified undelivered destination and its type (queue or topic) are added as new attributes to the notification.

This allows management applications to filter out notifications based on where the undelivered messages are preserved.

The following attributes are added to the **application.message.Undelivered** notification type:

- **IsUndeliveredDestinationQueue** — A value of **true** indicates that the undelivered message was preserved in a queue. A value of **false** indicates that the undelivered message was preserved in a topic.
- **UndeliveredDestination** — Name of the queue or topic where the undelivered message was preserved. If the message was preserved in the dead message queue, the value of this attribute is set to **SonicMQ.deadMessage**. If the message wasn't preserved, the value is blank.

Undelivered Destinations for DRA Messages

This section describes using the undelivered destination feature for messages that need to be delivered to a different routing node.

Undelivered Destinations Without a Node Name

If the specified undelivered destination is not a Temporary Topic, not a Temporary Queue, and it doesn't include a node name, when the message becomes undelivered it is preserved at the node where it became undelivered.

For example, if a queue message has expiration set and the message needs to be sent from node A to node B, the message may expire either at node A or at node B. Assume that the value of the undelivered destination property is the **SampleQ1** queue (note that there is no node name). If the message expires at node A, it is added to **SampleQ1** at node A. If it expires at node B, the message is added to **SampleQ1** at node B.

The same result is produced if the queue name is specified as **::SampleQ1**.

If the specified undelivered destination is a Temporary Topic or a Temporary Queue, when the message becomes undelivered, it is sent to the node where it was produced.

Undelivered Destinations With a Node Name

If an application needs to preserve its undelivered messages at a particular node, it can qualify the name of the undelivered destination with a node name.

Using the example from the previous section, the name "A::SampleQ1" ensures that undelivered messages is preserved in the queue SampleQ1 that resides at node A whether it has expired at node A or at node B.

However, in order for this to work, SampleQ1 must be declared global. Otherwise, if a message expires at node B, SonicMQ is not able to deliver it to SampleQ1 at node A. Instead, the message is delivered to node A and added to the system DMQ that resides at the first broker in node A that receives the message.

Required Routing Definitions

If undelivered destination includes an explicit node name, the specified node must be directly accessible from the node where the message becomes undelivered. Using the same example as in the previous section, node B must have a routing definition for node A. Otherwise the message is preserved in the system DMQ at the current broker in node B.

For example, if node B has a routing definition for node C which has a routing definition for node A, but node B has no routing definition for node A, the message is added to the system DMQ at the current broker in node B.

The same requirement exists for the undelivered destinations that specify temporary queues or topics. For example, if a message that specifies a temporary queue as its undelivered destination is sent from node A to node B and expires at some broker in node B, that broker attempts to send the message back to node A. However, if at the time of message expiration node B doesn't have a routing for node A, the message ends up in the system DMQ at the current broker.

Undelivered Message Reason Codes

Undelivered messages can result from routing of queue and topic messages, and from HTTP direct routing extensions. The reason codes, **JSM_SonicMQ_undeliveredReasonCode**, generated for these types of undelivered messages are described in this section. The reason codes are integers. The corresponding reason name is a **String** in **progress.message.jclient.Constants** that provides a description for the issue with the undelivered messages.

Table 33 lists the reason codes that relate to general cases of undelivered messages that can occur for both topic and queue messages, with or without dynamic routing or remote nodes.

Table 33: Reason Code for Undelivered Messages (Continued)

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).

Table 34 lists the reason codes that relate to messages whose delivery attempts to a receiver exceeded the specified maximum redelivery attempts.

Table 34: Reason Code for Undelivered Messages (Continued)

Reason	Value	Reason Marked as Undeliverable
UNDELIVERED_DELIVERY_LIMIT_EXCEEDED	28	MaxDeliveryCount was set to 1 or higher and that number attempts to deliver the message to the consumer have transpired without acknowledgement within the session. Message is discarded unless the message has the property SonicMQ_preserveUndelivered set to true.

Table 35 contains a reason code that can occur only for undelivered queue messages under dynamic routing. See the chapter “Multiple Nodes and Dynamic Routing” in the *Aurea SonicMQ Deployment Guide* for some examples of dynamic routing of queue messages.

Table 35: Reason Codes for Undelivered Queue Routing Messages (Continued)

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 36 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 chapter “Multiple Nodes and Dynamic Routing” in the *Aurea SonicMQ Deployment Guide* for some examples of dynamic routing of queue messages and examples of remote publishing and subscribing.

Table 36: Reason Codes for Undelivered Routing Messages (All Domains) (Continued)

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.

Reason	Value	Reason Marked as Undeliverable
UNDELIVERED_ROUTING_CONNECTION_AUTHENTICATION_FAILURE	7	Routing connection username and password were not authorized at a routing node while connecting to the remote broker.
UNDELIVERED_ROUTING_CONNECTION_AUTHORIZATION_FAILURE	8	Routing connection username did not have appropriate permissions to connect to the specified routing node. (Route ACL)
UNDELIVERED_MESSAGE_TOO_LARGE_FOR_QUEUE	9	Message is larger than the size of the queue.
UNDELIVERED_ROUTING_MULTI_TOPICS_NOT_SUPPORTED	27	A MultiTopic publish was attempted to a remote node that does not support multitopic PubSub—for example, a V6.0 broker.

Table 37 lists the reason codes that occur only for undelivered topic messages. See the chapter “Multiple Nodes and Dynamic Routing” in the *Aurea SonicMQ Deployment Guide* for some examples of remote publishing of topic messages and remote subscribing to topic messages.

Table 37: Reason Codes for Undelivered Topic Routing Messages (Continued)

Reason	Value	Reason Marked as Undeliverable
UNDELIVERED_ROUTING_TOPIC_MESSAGES_NOT_SUPPORTED	18	Message could not be delivered to the destination because the remote node does not support remote topic messages.
UNDELIVERED_ROUTING_SUBSCRIPTION_AUTHORIZATION_FAILURE	19	Subscription request could not be delivered to the destination because the remote node denies subscribe permission to the routing user.
UNDELIVERED_ROUTING_REMOTE_SUBSCRIPTION_DELETED	20	Message has been marked undelivered because the remote subscription was deleted or has expired.
UNDELIVERED_ROUTING_REMOTE_SUBSCRIPTIONS_NOT_SUPPORTED	21	Subscription request could not be delivered to the destination because the remote node doesn't support remote subscriptions.

Table 38 lists the reason codes that can occur for undelivered HTTP Direct routing messages. See the chapter “HTTP Direct Acceptors and Routings” in the *Aurea SonicMQ Deployment Guide* for information about HTTP Direct routing and examples of how to implement it in your deployments.

Table 38: Reason Codes for Undelivered HTTP Direct Routing Messages (Continued)

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 has been 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.
UNDELIVERED_JMS_QUEUE_NOT_FOUND	22	A replyTo destination is not found when processing a reply in a HTTP Direct Outbound scenario.

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 39 might occur.

Table 39: 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 has been marked undelivered because the request was sent to an unregistered URL (in other words, there is no protocol handler listening for requests on that URL).
UNDELIVERED_HTTP_HOST_UNREACHABLE	11	Message intended for dynamic routing over HTTP has been marked undelivered for one of the following reasons: A connection cannot be made to the HTTP destination The request has timed out

Table 40 contains reason code that can occur for undelivered messages when an undelivered destination has been defined by the user application. See [Specifying a Destination for Undelivered Messages](#) on page 296.

Table 40: Reason Codes That Relate To a User-Specified Dead Message Queue (Continued)

Reason	Value	Reason Marked as Undeliverable
UNDELIVERED_JMS_QUEUE_NOT_FOUND	22	The undelivered destination is a queue and that queue does not exist.
UNDELIVERED_QUEUE_FULL	23	The undelivered destination is a queue and that queue is full. Note that if the undelivered destination is a queue and the size of the message exceeds the max size of that queue the reason code is <code>UNDELIVERED_MESSAGE_TOO_LARGE_FOR_QUEUE</code> .
UNDELIVERED_TOPIC_FULL	24	The undelivered destination is a topic and at least one of the subscribers for that topic is flow controlling its publishers. Note that even if a subscriber uses <code>FlowToDisk</code> , it may flow control its publishers (for example, if the maximum topic database size has been exceeded).
UNDELIVERED_UNSUPPORTED_OVERRIDE_DESTINATION	25	The undelivered destination is an HTTP Direct destination.
UNDELIVERED_INVALID_PROPERTY_TYPE	26	When the undelivered destination is not set by a compatible client application using a correct setter method, this error could occur.

Note: If the undelivered destination resides in a different routing node and the message can't be delivered there because of a DRA-related problem, the reason code is one of the DRA-related codes in the `progress.message.jclient.Constants` class.

Recoverable File Channels

This chapter explains the recoverable file channel feature available with SonicMQ installations containing ClientPlus.

For details, see the following topics:

- [About Recoverable File Channels for Large Messages](#)
- [Tips and Techniques for Using File Channels](#)

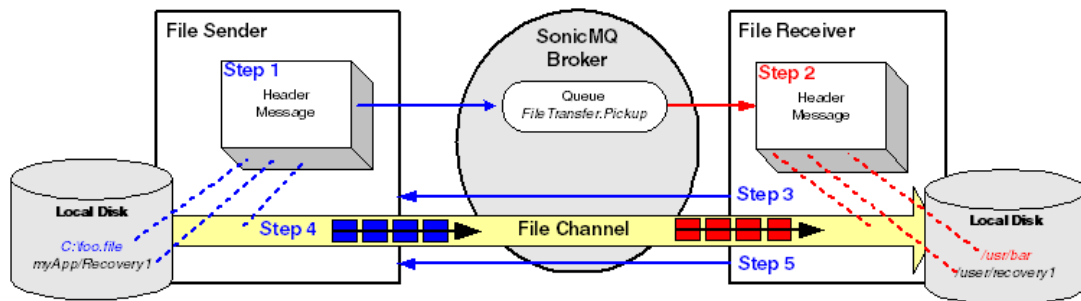
About Recoverable File Channels for Large Messages

SonicMQ® installations that provide ClientPlus features have the option of using recoverable file channels for very large messages—messages larger than 10 megabytes—where the data content is stored in a file on the sender system. The receiver of such a message would store the large message in a file on its system. Handling a very large message as a single integral object puts a strain on system resources such as broker and client memory.

SonicMQ® provides Point-to-point domains with a **RecoverableFileChannel** that is attached to a standard JMS message. The receiver of such a message retrieves and performs operations on the **RecoverableFileChannel**.

A **RecoverableFileChannel** is a unidirectional stream of information from a JMS client to a JMS client, a peer-to-peer type of transfer, similar to FTP. State and recovery information is stored on both the sending and receiving clients and is separate from state and recovery information stored on brokers.

Figure 70: Large Message Moving in Fragments Across a File Channel



The above figure illustrates the basic steps in a file transfer:

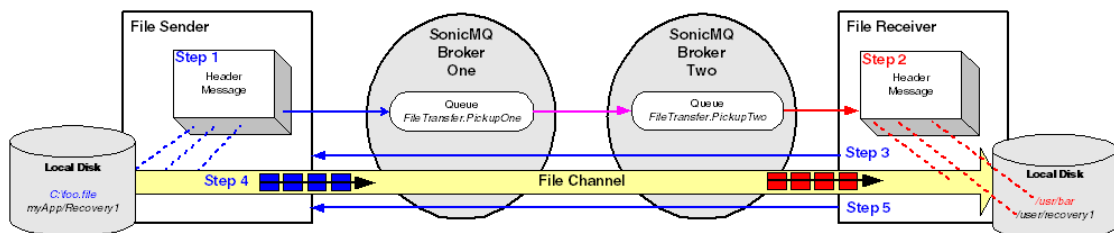
1. The file sender creates a header message, gets an instance of a file channel, identifies the file to transfer and then sends the header to the SonicMQ® broker queue where receivers listen (shown as, **FileTransfer.Pickup**).
2. The file receiver takes the header message of the queue and prepares a file target location and a recovery log on its local disk.
3. The receiver indicates that it is ready to continue (start) the transfer. At this point, the header message has completed its purpose and is discarded.
4. The sender fragments and packages a portion of the source file. The sender will continue to do this until it reaches its defined window size at which point it waits for acknowledgement.
5. When the transfer completes successfully the receiver lets the sender know that the transfer is done.

Forwarding the Header Message

Receiving the header message does not obligate the initial receiver to participate in the actual file transfer. The header message could be forwarded to another queue on another broker where another receiver can take the message. The requirement is that the final receiver—the one that intends to consume the large message by managing the channel delivery—must be able to access global queues on the sender's broker. The channel is acted on by explicit acknowledgement followed by a call to **Message.getChannel()** to retrieve the **RecoverableFileChannel** reference.

The forwarding application must take care that it takes steps to ensure that it doesn't forward the message and then get the channel. The following figure illustrates the sequence of events in a file transfer across global queues.

Figure 71: File Transfer Across Accessible Global Queues on Two Brokers



Global Queues

When a connection is dropped you might want to establish a **QueueConnection** on a different broker and recover the transfer. This is a good tactic when you are using the Dynamic Routing Architecture provided that the broker is part of the same routing node.

If you connect to a different routing node, the transfer will not continue successfully.

Dynamic Routing Architecture

SonicMQ's Dynamic Routing Architecture can use recoverable file channels when both peer applications can access global queues of the other client's broker.

Semantics of File Fragmentation, Transfer, and Recovery

The mechanics of disassembling and reassembling files such that a transfer can be resumed within the file require some semantic concepts. These methods involve primarily to the **Channel** interface. Terms used in these methods include:

Timeout

When a JMS message is sent with a **RecoverableFileChannel** attached, the send call blocks until a channel is established. This behavior helps manage channels because the sender knows at send time whether or not a receiver has accepted the channel. If a channel is not established in **Timeout** time, a **JMSEException** is thrown with the error code:

```
progress.message.jclient.ErrorCodes.ERR_CHANNEL_TIMEOUT.
```

Retry Count and Retry Interval

When a fragment is sent, an acknowledgement is expected from the receiving client. If an acknowledgement is not received in **retryInterval** time, the fragment is resent. This continues **retryCount** times, at which time the **ChannelListener** callback is called with the **RFC_RETRY_TIMEOUT** error code.

These commands take effect immediately when called from within the **ChannelListener** callback, when set before **completeConnect()** is executed, or before the channel is sent. While these values can be set from outside the **ChannelListener** callback, they will not take effect until the underlying code notices the change.

On the receiver, these methods can be used to timeout internal sender/receiver communication:

- If a fragment is not received in **retryCount * retryInterval** time, the receiver's **ChannelListener** callback is executed.
- The retry count and retry interval values are used when there is a call to **completeConnect()** on the receiver. If the sender does not respond, the receiver attempts to contact the sender **retryCount** times, waiting for **retryInterval** time for an acknowledgement. After that time, the **completeConnect()** call throws a **JMSEException** with the error code **progress.message.jclient.ErrorCodes.ERR_RETRY_TIMEOUT**.
- During a recovery, if the other side of a recovery is unavailable, a **JMSEException** is thrown from the **continueTransfer()** call with the error code, **progress.message.jclient.ErrorCodes.ERR_RETRY_TIMEOUT**.

Fragment Size

When the transfer breaks a large file into fragments, the fragments are sent, each expecting an acknowledgement from the receiving client. Performance can be optimized by configuring the fragment size. The larger the fragment size, the more memory needed on the client and broker to transfer the message.

To determine an optimal fragment size setting, first determine the actual TCPIP packet size for your environment, then make the fragment size a direct multiple of the TCPIP packet size. For example, if the packet size is 512 bytes, make the fragment size 512, 1024, or another multiple of 512 bytes. This method of determining fragment size optimizes the amount of breakage of the message that is sent. For example, if the packet size were 512 bytes and you set the fragment size to 768 bytes, you send 33% more packets than needed. This setting fills one packet to 512 bytes and the next to only 256, wasting the remaining 256 bytes of that packet.

Window Size

The window size is the number of fragments that can be sent before an acknowledgement must be received. The larger the window size, the more time will elapse before an acknowledgement is required. A larger window size is preferred for messaging setups with a high degree of latency between the sender and receiver. A larger window size could also cause the sender to take more time before noticing the receiver is unavailable. A smaller window size could cause the sending of the file to halt too often to wait for acknowledgements from the receiving client.

ChannelID

Every channel has a unique channel ID that is assigned to it when the channel is established. While this value is primarily for internal purposes, the channel ID is accessible by the user to help associate information with a channel ID. The channel ID is available across failures. The **channelID** is null until after the channel has been sent, or until **getChannel** has been called on the receiver.

Canceling or Closing a Channel

When you cancel a transfer, you delete all recovery information.

When you close a transfer, you just stop the channel transfer. All recovery information is preserved.

A call to **cancel()** or **close()** notifies the other side of the transfer that the channel has been cancelled or closed. However, a broker or client failure could cause this notification not to be delivered. If this occurs, the other side of the transfer eventually times out after **retryCount*retryInterval** time. Then the application should cancel the transfer and resend the header message.

Classes and Interfaces for Large Message Transfers

The classes and interfaces to support large messages are the following:

Class:

- **ConnectionFactory** — The method that is essential to client local persistence is also used with **RecoverableFileChannels** to set a local store directory for recovery information for the channel. If this method is not set, the working directory is used to store recovery information
- **RecoverableFileChannelFactory** — The constructor class for the recoverable file channel in the method:

```
static RecoverableFileChannel createRecoverableFileChannel(java.io.File
file)
```

Interfaces: (progress.message.jclient)

- **Message** — Added methods for setting and getting channels on a message.
- **QueueConnection** — Added methods for getting reference to channels and working with unfinished channels.
- **Channel** — Encapsulates the control and recovery logic for sending a message data stream. A sender application instantiates an implementation of this class to send a large message.
- **channel.RecoverableFileChannelFactory** — Creates recoverable file channels.
- **channel.RecoverableFileChannel** — A **Channel** implementation to send files on a channel.
- **ChannelListener** — Defines a method for notifying a sender or receiver application that a large message transmission has completed successfully.
- **ChannelStatus** — Enables queries to a channel for its status.

The methods that are relevant to recoverable file channels are listed in the following table.

Table 41: Methods that Support Recoverable File Channels (Continued)

Interface	Method	Description
Message	void setChannel(Channel <i>channel</i>)	Adds <i>channel</i> to the message.
	void setChannel(Message <i>message</i>)	Adds the channel associated with <i>message</i> to this channel. This method provides a technique for forwarding a message to another destination with a defined channel attached to a new message.
	boolean <i>hasChannel</i> ()	Returns true if this message has a channel attached.
	Channel <i>getChannel</i> ()	Retrieves the channel attached to this message.
QueueConnection	Channel getChannel(String <i>channelID</i>)	Returns the channel <i>channelID</i> .
	boolean <i>hasUnfinishedChannels</i> ()	Tests whether the channel has any unfinished channels.
	Enumeration <i>getUnfinishedChannels</i> ()	Returns an Enumeration containing Channel references which can be cast to a RecoverableFileChannel. You can iterate over the enumeration, checking channel status, and recovering channels.
	Enumeration <i>getUnfinishedChannelsIDs</i> ()	Returns an Enumeration of Strings representing the ChannelIDs that can retrieve specific channels with the getChannel(String <i>channelID</i>) method.
	boolean <i>hasUnfinishedChannel</i> (String <i>channelID</i>)	Tests whether the channel <i>channelID</i> exists.

Interface	Method	Description
ConnectionFactory	void setLocalStoreDirectory (String <i>name</i>)	Sets a local store directory that to store the recovery information for RecoverableFileChannel.
Channel	void setTimeout (long milliseconds)	Sets the timeout time of this channel to <i>milliseconds</i> . Default is 60000— one minute.
	void setRetryCount(int <i>count</i>)	Sets the <i>count</i> of times a fragment of information will attempt to be sent before timing out. Default is 10 attempts.
	void setRetryInterval (long milliseconds)	Sets the retry interval <i>milliseconds</i> to wait before attempting to retry to send or receive a fragment. Default is 10000— ten seconds.
	void setFragmentSize(int <i>size</i>)	Sets the <i>size</i> in bytes for each fragment to transfer on this channel. Default is 1024 bytes.
	void setWindowSize(int <i>size</i>)	Sets the number of fragments to buffer to <i>size</i> —an integer value of 3 or higher—to allow that many fragments to await acknowledgement before blocking the send. Default is 10 fragments.
	void setUUID (java.lang.String <i>uuid</i> , int <i>timeToLive</i>)	Sets the duplicate detection <i>uuid</i> —universal unique id—to activate duplicate detection for the channel. <i>timeToLive</i> (in minutes) indicates how long this UUID is reserved.
	void setChannelListener (ChannelListener <i>listener</i>)	Sets channel listener <i>listener</i> on the channel.
	void continueTransfer()	Continues the transfer of information between the sending and receiving client.
	java.lang.String getChannelID()	Gets the channelID associated with this channel.
	long getTimeout()	Gets the timeout time of the channel.

Interface	Method	Description
Channel (continued)	int getRetryCount()	Gets the retry count.
	long getRetryInterval()	Gets retry interval.
	int getWindowSize()	Gets the window size for the transfer.
	int getFragmentSize()	Gets the size of the fragments on this channel.
	ChannelStatus getChannelStatus()	Gets the current channel status. Returns a ChannelStatus object that indicates the current status of the channel.
	void completeConnect()	Completes the connection with the sender. Starts the data transfer on the channel.
	void close()	Closes the channel and stops the channel transfer. Recovery information is preserved.
	void completeTransfer()	Completes the transfer of the file. This method is called from the ChannelListener when the transfer is done.
RecoverableFileChannel	void save(java.io.File file)	Saves the file associated with this channel to file.
	boolean isSaving()	Tests whether this file channel is actively saving information to a file.
	void cancel()	Cancels the channel. Closes the channel and removes any recovery information.
	void setBlockSize(int size)	Sets the write buffer block size to <i>size</i> bytes. Default is <code>windowSize * fragmentSize</code> , which is, when those parameters are also defaulted, 10240 bytes. When security is enabled, the actual value used is the value entered rounded to the next 8 KB.
RecoverableFileChannel (continued)	void setDES (java.security.Key key)	Set the channel to read and write the file as DES encrypted with <i>key</i> .
	void setDESede (java.security.Key key)	Set the channel to read and write the file as Triple DES encrypted with <i>key</i> .
ChannelListener	void onChannelStatus (Channel channel, java.lang.Exception e)	This is called by the channel whenever a status change occurs in the channel.
ChannelStatus	long bytesTransferred()	Return the number of bytes that have been transferred on this channel. The value -1 is returned if unknown.

Interface	Method	Description
	long bytesToTransfer()	Return the number of bytes to transfer. The value -1 is returned if unknown.
	int getStatus()	Return the current status of the channel. See Channel Status on page 314.

ChannelListener

A channel listener is set on a channel to inform an application asynchronously of status and error conditions. If there is a problem transferring a file such as a timeout and there is no **ChannelListener** or the program logic does not perform a **continueTransfer** in the channel listener, the channel is implicitly cancelled in which case the **QueueConnection'sonException** listener is called with **progress.message.jclient.ErrorCodes.ERR_CHANNEL_IMPLICITLY_CANCELLED**.

continueTransfer() in the channel listener is valid only when the status is **RFC_RETRY_TIMEOUT**. This means an application can increase the value of **RetryCount** or **RetryInterval** so that after (count * interval) milliseconds has passed without receiving any acknowledgement, the channel listener is called again. Any retries will be performed after the **ChannelListener** returns. You might choose instead to close or cancel the channel.

The **onChannelStatus** method is called with a reference to the **Channel** in question, and an Exception **e** that triggered this channel status call.

Important: Thread Safety considerations

While an application should manipulate a channel while inside a **ChannelListener**, the application can safely call a number of methods from outside the **ChannelListener** thread without error: **Channel.close()**, **Channel.getChannelStatus()** and any method of the **ChannelStatus** object like **RecoverableFileChannel.cancel()** and **RecoverableFileChannel.isSaving()**.

The following methods can be called from outside the **ChannelListener** callback but they might not take effect immediately: **Channel.setRetryCount()**, **Channel.setRetryInterval()**.

All other methods should not be considered thread safe and should only be called within a **ChannelListener** or before the channel is established.

Channel Status

A channel status object is a user's window into the current status of a channel. The channel status object is retrieved by a **getChannelStatus** call on the **progress.message.jclient.Channel** interface. The current channel state can be retrieved by the **channel.getChannelStatus().getStatus()** method, then the application should use a switch statement on the status code to set the course of action. Many channel commands are not thread safe or are not valid when a channel is in a given state; therefore, it is recommended that the developer put the logic that affects the transfer of the channel within the **ChannelListener**. The **ChannelStatus.getStatus** indicates the status of the file transfer as listed in the following table.

Table 42: Error Codes Returned for Channel Status (Continued)

Error Constant	Meaning
RFC_TRANSFERRING	The file is currently transferring.
RFC_TRANSFER_COMPLETE	The file transfer is complete.

Error Constant	Meaning
RFC_CANCELLED	<p>The file transfer was cancelled. This informative status indicates that recovery information has been erased.</p> <hr/> <p>Note: The ChannelListener on a local client is not called if cancel was called locally.</p> <hr/>
RFC_CLOSED	<p>The file transfer was closed. This informative status indicates that local recovery information has not been erased so the channel can be recovered from a queue connection at a later time.</p> <hr/> <p>Note: The ChannelListener on a local client is not called if close was called locally.</p> <hr/>
RFC_WAITING_CONNECT	The file transfer is waiting to connect to the remote client.
RFC_RETRY_TIMEOUT	<p>The file transfer is currently timed out. Method calls at this time include:</p> <ul style="list-style-type: none"> • continueTransfer to continue trying to contact the remote client for retryCount attempts at retryInterval. • close to stop the transfer, close the channel, and retain recovery information. • cancel to stop the transfer, close the channel, and erase recovery information. • setRetryCount to change the retry count on the next continueTransfer. • setRetryInterval to change the retry interval on the next continueTransfer.
RFC_PRECONNECT_ATTEMPT	The file channel has been restored but continueTransfer() has not been called.
RFC_DISCONNECT	The QueueConnection to the broker has disconnected.
RFC_LOCAL_ERROR	There has been a fatal error—such as an IOException while attempting to access the file—on the local client. Examine the Exception that is passed in to the ChannelListener to determine the problem. The channel is closed, and the recovery information is retained.
RFC_REMOTE_ERROR	There has been a fatal error—such as running out of disk space—on the remote client. Examine the Exception that is passed in to the ChannelListener to determine the problem. The channel is closed, and the recovery information is retained.

General Procedure for Large Message Transfers

The establishment and use of channels is defined by the sending and receiving JMS clients. The general steps for using a channel are as follows:

1. The sender:
 - a. Creates a **QueueConnection** where client persistence is not enabled.
 - b. Creates a **nontransactedQueueSession**.
 - c. Creates a **QueueSender** to a queue (local or remote).
 - d. Creates a JMS message of any type.
 - e. Instantiates a **RecoverableFileChannel**.
 - f. Calls **Message.setChannel(RecoverableFileChannel)**.
 - g. Sends the message with the channel instance.
2. The receiver:
 - a. Creates a **QueueConnection**.
 - b. Creates a **QueueSession** in **CLIENT_ACKNOWLEDGE**, or **SINGLE_MESSAGE_ACKNOWLEDGE** mode. The message could be in a transacted session; however, you can only get the channel once the session **commit** is called, an explicit acknowledgement.
 - c. Creates a **QueueReceiver** to the queue where the message with the channel instance is available.
 - d. The receiver takes and acknowledges—**acknowledge()** or **commit()** when transacted—the JMS Message.

Important: The acceptable acknowledgement modes require explicit acknowledgement by calling **Message.acknowledge()**. You can not get a channel from a message that is delivered on a session with **AUTO_ACKNOWLEDGE** or **DUPS_OK_ACKNOWLEDGE**.

1. The receiver makes a call to **Message.getChannel()** to retrieve the **RecoverableFileChannel** reference. This returns a reference to a **Channel** that can be cast to a **RecoverableFileChannel**.
2. The receiver sets options for the transfer—such as **RetryInterval** and **RetryCount**—and a callback.
3. A call is made to **Channel.completeConnect()**. The transfer of data from the sender to the receiver starts in the channel.
4. A call is made to **RecoverableFileChannel.save(File file)**.

Note: The Large Message mechanism is not recommended as the only communication in an application. It is optimized for reliability across failures and is not the best solution for messages that can be transferred as a single JMS message. Large Message transfers involve protocol overhead and disk I/O that would not be incurred in a normal JMS Message.

Creating a Recoverable File Channel

In the following excerpt from **FileSender.java**, a recoverable file channel is created bound to a designated file and a unique identifier. The channel is added to any type of JMS message by a **setChannel** method. The message property **SenderFileName** is set with the absolute location of the disk-based file that is intended for transfer (for example, `c:\uploads\today.data`). With everything ready, the header message—the message with data about the file channel—is sent.

When a receiver acknowledges a message that has a channel, the receiver application calls **getChannel** to retrieve the **Channel** reference. The receiver calls **completeConnect()** to get the transfer underway. The data transfers from the sender to the receiver without any explicit action required in the sender JMS send session or the receiver JMS receive session. The channel listeners and the channel status provide the information for the recovery logs as the transfer proceeds.

You can tune the transfer by using setters for the retry count, the retry interval (in milliseconds), the message fragment size, and the window size—the number of fragments (>3) that can be sent before the sender blocks to await acknowledgement from the receiver. You might also want to use the **ChannelStatus** methods of **bytesTransferred** and **bytesToTransfer** to estimate completion time.

The following code snippet shows the **sendNewMsg** pattern in the **FileSender** sample.

FileSender Sample: sendNewMessage Pattern (Continued)

```
//send new msg with a channel
try
{
    javax.jms.TextMessage msg = (javax.jms.TextMessage)session.createTextMessage(
    "Test");
    //create RecoverableFileChannel object and set ChannelListener to it
    progress.message.jclient.channel.RecoverableFileChannel rfc =
    progress.message.jclient.
        channel.RecoverableFileChannelFactory.createRecoverableFileChannel(file);

    rfc.setChannelListener(new LMSChannelListener());
    //set Timeout to a channel, so if during 60sec the channel is not established,

    //ChannelListener will cancel it
    rfc.setTimeout( 60000);

    //set a channel to a message
    ((progress.message.jclient.Message)msg).setChannel( rfc );

    //set message property (name of a file being sent), which is used by receiver
    //to get file name
    msg.setStringProperty( "SenderFileName", file.getName() );
    //send message
    System.out.println( "\nTry to send header message and establish channel to send
    file - "
        + file.getAbsolutePath() );
    sender.send( msg );
    System.out.println( rfc.getChannelID() + " channel established!" );
    //begin to browse the transfer progress
    browseTransfer(rfc);
} catch( javax.jms.JMSEException jmse )
{
    System.out.println( jmse.getMessage() );
} catch( Exception e )
{
    System.out.println( e.getMessage() );
}
```

Recovering an Interrupted Transfer

Each **RecoverableFileChannel** maintains a recovery file to ensure that it can be restarted in the event of the interruption before completion. Fragments could be dropped during the transfer due to broker failure or invocation of flow control, the recovery handles the retransmission and ordering of message fragments.

Methods in the **QueueConnection** class initiate recovery. An application uses the **hasUnfinishedChannels** method to check for recovery log files. If there are incomplete transfers, the application can iterate through the channels, calling **continueTransfer()** to reestablish the channel and continue the transfer from the last logged fragment.

Important: If a system failure occurs in the early stage of the transfer—within the channel connection negotiation, before any information has transferred—it is possible for the recovery logs to be corrupt. If the recovery logs are corrupt, **hasUnfinishedChannels()** or **hasUnfinishedChannel(String channelId)** might return **true**, yet **getUnfinishedChannel(String channelId)** could return null.

Patterns for Recovery

The code samples in this section demonstrate two techniques for handling unfinished channels:

- The sender gets the unfinished channels as an enumeration then iterates through the list. For each unfinished channel, it sets up a listener then continues the transfer.
- The receiver gets the channel identifiers, using each one to try to get its channel reference. When it is successful, it sets up the listener, continues the transfers and resumes the file-save procedures. This technique lets you associate additional information with a channel, as demonstrated in the **FileReceiver** sample where it associates the filename.

The following code snippet shows the **RestoreChannels** pattern in the **FileSender** sample.

FileSender Sample: RestoreChannels Pattern

```
private void restoreChannels()
{ try
{

// Check if there are any unfinished channels to restore
if( connection.hasUnfinishedChannels() )
{

// Retrieve an enumeration of all available Channels of this client
Enumeration channels = connection.getUnfinishedChannels();
while( channels.hasMoreElements() )
{
progress.message.jclient.channel.RecoverableFileChannel channel = null;
channel = (progress.message.jclient.
    channel.RecoverableFileChannel) channels.nextElement();
try
{

// Set a channel listener
channel.setChannelListener( new LMSChannelListener() );
System.out.println("\nTry to restore channel - " + channel.getChannelID());

// Continue the transfer
channel.continueTransfer();

//begin to browse the transfer progress
```

```
browseTransfer(channel);
}
catch( javax.jms.JMSEException jmse)
{
channel.cancel();
System.out.println( jmse.getMessage() );
}
}
}
}
catch( javax.jms.JMSEException e )
{
System.out.println( e.getMessage() );
}
catch( IOException e )
{
e.printStackTrace();
}
}
```

The **RestoreChannels** pattern in the **FileReceiver**, shown in the following code snippet, is slightly different from the **FileSender**.

FileReceiver Sample: RestoreChannels Pattern

```
private void restoreChannels()
{ // First clean the property log file. There can be extra records

// ( see comments in onMessage() )
cleanPropFile();

try
{

// Check if there are any unfinished channels to restore
if( connection.hasUnfinishedChannels() )
{

// Retrieve an enumeration of Strings contains all available channelIDs
Enumeration channelIDs = connection.getUnfinishedChannelIDs();
while( channelIDs.hasMoreElements() )
{
String channelID = null;
progress.message.jcclient.channel.RecoverableFileChannel channel = null;
String channelFile = null;

// Get the channelID
channelID = (String)channelIDs.nextElement();

// Retrieve the channel reference. If failed try the next element
channel =
(progress.message.jcclient.channel.RecoverableFileChannel)connection.getChannel(
channelID );
try
{

// Set a channel listener
channel.setChannelListener( new LMSChannelListener() );

// Access our properties to print useful information
channelFile = saveDirName + File.separator + msgProp.getProperty( channelID );

System.out.println( "\nTry to restore channel -" + channelID + "- and save into
file - " + channelFile );

// Continue the transfer
channel.continueTransfer();

//Check if channel was being saved before failure. If not, then we
//specify the file - channelFile (from property log file three strings above) -
```

```
//where to save received fragments
if( !channel.isSaving() )
channel.save( new File( channelFile ) );
}
catch( javax.jms.JMSEException jmse )
{
channel.cancel();
System.out.println( jmse.getMessage() );
}
}
}
...
}
```

Duplicate Detection for File Transfers

SonicMQ provides a mechanism for detection of duplicate messages to ensure that a message is not sent more than once. You enable duplicate detection on file channels with the **Channel** method **setUUID(String uuid, int timeToLive)**.

The nature of large message transfers makes it important to test for duplicates at the inception of the transfer. When a fatal error occurs during an ongoing file transfer, it might be practical to have the entire file resent by a different client. But the application should prevent two clients from sending the same large message at the same time.

When duplicate detection is set, a 32-character universally unique ID (UUID) is assigned that relates to the message being sent. The **timeToLive** parameter of the method specifies how long a channel transfer can be inactive before another channel with the same UUID can be reused. After the transfer has completed, the UUID cannot be reused for a new channel.

To distinguish between when the channel is to be active and when the channel has been duplicated, two exceptions are defined:

- **ChannelDuplicateDetectException** is thrown if there has been a duplicate of a message.
- **ChannelActiveException** is thrown if the channel is perceived to be active. This exception has a **getTime()** method that returns when the channel might be inactive.

See the [Duplicate Message Detection Overview](#) for more information.

Note: Duplicate detection is normally a sender issue. For multi-broker architectures, the UUID registration is therefore resident on the sender's broker. But file transfers favor the receiver watching for duplicates so the UUID registration for file transfers resides on the receiver's broker.

An UUID-related exception can occur asynchronously during a transfer, **ERR_CHANNEL_UUID_IN_USE**. This error can happen when the receiver, continually updating the duplicate detection persistent storage mechanism on its broker, gets stressed such that it cannot keep up notification that it is active. If another channel starts with that same UUID, the error is thrown. You can minimize the chance of this occurring by setting the **timeToLive** value on the UUID to a higher value. Rather than **1** (one minute), try **10** to allow the systems to loosen up.

Security on File Transfers

Encryption can be used for file transfers so that is read and written DES or Triple DES encryption using the key provided. The setter, **setDES(java.security.Key key)** or **setDESede(java.security.Key key)**, is called before the channel is established between the sending and receiving clients—before the header message is sent, before **completeConnect()** is called by the receiver, and before **continueTransfer()** is called on recovery.

The sender's key and receiver's key are unrelated. The sender can send an unencrypted file that the receiver encrypts before saving the file. Specifically:

- Setting DES or DESede on the sender implies that the local file is to be read using the encryption type and key provided.
- Setting DES or DESede on the receiver implies that the file that is written on the receiver system will use the stated encryption type and key provided.

Java Cryptography Extension (JCE), accessible from Sun's Web site, is an important part of DES encryption and some JVMs provide it while others do not. The Java Runtime Environment (JRE) bundled with SonicMQ for installations under Windows NT includes IBM's JCE.

Using Multiple File Channels

When one client is using channels, the client might need to recover multiple channels. The recovery method, **continueTransfer()**, blocks and times out when the sending or receiving client is not available to continue the transfer.

Important: When several channels might be in use, **do not let two or more clients share a local store directory** as the recovery information could be corrupted for all the clients sharing the same local store directory.

Methods for handling unfinished channels are provided on the **QueueConnection** to manage and iterate through unfinished channels.

See the recovery patterns in code excerpts from the samples at [Recovering an Interrupted Transfer](#) on page 318.

Exception Handling for File Channels

When designing applications with recoverable file channels, create logic to handle common error situations, which include:

- On the **QueueSender.send()** call
- On the **Channel.completeConnect()** call
- During recovery, on the **Channel.continueTransfer** call
- During an active file transfer

The **Channel** implementation provides **ErrorCodes** that isolate problem cases. These error codes are listed in the following table.

Table 43: Error Codes from the Channel Implementation (Continued)

Error Constant	Meaning
ERR_CHANNEL_TIMEOUT	There has been a time out while attempting to establish a channel.
ERR_CHANNEL_RETRY_TIMEOUT	There has been a time out while attempting to transfer the file.
ERR_CHANNEL_RECOVER_FILE_UNREADABLE	The file to transfer is unreadable.
ERR_CHANNEL_IMPLICITLY_CANCELLED	There is no channel listener and the channel was canceled.
ERR_CHANNEL_INTERNAL_ERROR	There has been an internal JMS Exception. Call <code>getLinkedException()</code> to determine the problem.
ERR_CHANNEL_IO_ERROR	There has been an internal IO Exception. Call <code>getLinkedException()</code> to determine the problem.
ERR_CHANNEL_ALREADY_ESTABLISHED	An attempt was made to establish a channel with a sender that already has a channel established.
ERR_CHANNEL_ACTIVE	The channel is currently active. This is the error code used with the <code>ChannelActiveException</code> .
ERR_CHANNEL_DUPLICATE	The channel has been duplicated. This is the error code used with the <code>Channel Duplicate Detect Exception</code> .
ERR_CHANNEL_UUID_IN_USE	Another channel successfully established a channel using the UUID specified for this channel. This might occur when applications are under heavy load and the channel was seen as inactive. The <code>timeToLive</code> value, <code>ttl</code> , in the <code>setUUID(String uuid, int ttl)</code> method might be too small. See Duplicate Detection for File Transfers on page 320 for more information.
ERR_CHANNEL_FATAL_DUP_DETECT_EXCEPTION	There was a fatal error while accessing duplicate detection.
ERR_CHANNEL_TRANSFER_CLOSED	The transfer was closed. Existing recovery information is preserved.
ERR_CHANNEL_INVALID_KEY_TYPE	The type of key presented for encryption/decryption was not the type expected.
ERR_CHANNEL_INVALID_DECRYPTION_KEY	The key presented for encryption/decryption was not valid.
ERR_CHANNEL_JCE_UNAVAILABLE	The Java Cryptography Extensions were required by <code>setDES</code> or <code>setDESede</code> and were not available.

Log Files

Log files are produced on both the sender and the receiver when a file transfer starts. These logs enable recovery. Both the sender and the receiver use their logs to re-establish contact. As such, if one of the participants in file transfers prefers to cancel unfinished channels while the other intends to complete them, the logs do not exist to complete recovery. The identity of the channel and the bytes transferred enable the receiver to specify the point where recovery should resume.

These files and folders should never be shared. Two applications on the same system should provide distinct folders. Note that the default folder name is the name of the current working directory. You should make efforts to be more emphatic. For example, the samples in **ClientPlus/LargeMessageSupport** are two applications in the same folder. The user name provided in the startup of each application is assigned to the folder name for the application so that the two applications do not share. You could hardwire a very specific name into every sender and receiver application to ensure that recovery logs are distinct.

Tips and Techniques for Using File Channels

Some design and logic techniques make file transfers easier to work with, such as:

- Do not share local store directories across applications. If you do explicitly set the local store directory, the application's working directory is used to store recovery log files. When multiple channel applications might run in that working directory, recovery files can be damaged.
- The ChannelListener is only informed during a file transfer. The listener is tuned in to the channel, not the sender and receiver sessions.
- While it is generally advised, all brokers within a routing node must have unique names to use channels.
- You cannot enable client persistence mechanisms (see [Client Persistence](#)) in the same connection that you intend to use recoverable file channels.
- The maximum file size supported is dependent the Java Virtual Machines running on the participants' systems. The general limit on JVM's earlier than 1.2 is 2 Gigabytes.
- If a **send()** call times out, the header message can still be available on queues but the receiver will experience an **ERR_RETRY_TIMEOUT** exception when it tries to establish the channel. It is practical to set the **timeToLive** to the same value as the timeout of the channel.
- Directory and files that are created on the sender and the receiver enable recovery. If you delete the files or folders on either the sender or receiver, the other peer might attempt recovery but it will not be achievable.

SonicStream API

Important: This feature requires the SonicMQ installations of the participants in a SonicStream transfer—the sending application, the broker, and the receiving applications—are installed, upgraded, or updated to SonicMQ 2013 and later versions.

For details, see the following topics:

- [About the SonicStream API](#)
- [Common SonicStreamFactory Semantics](#)
- [Stream Publisher Semantics](#)
- [Stream Subscriber Semantics](#)
- [Managing Flow Control](#)
- [Handling Errors](#)
- [Samples of SonicStreams](#)

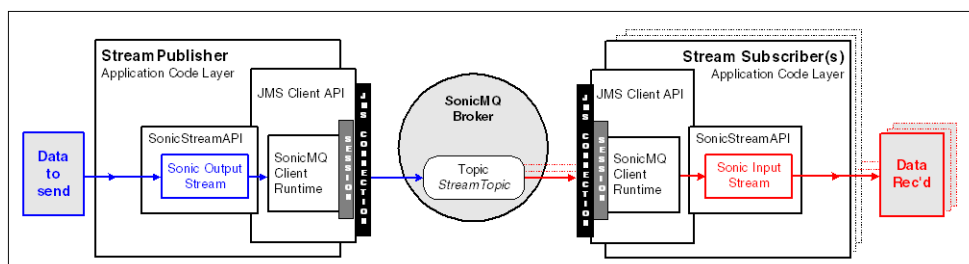
About the SonicStream API

SonicMQ® installations have the option of using the **SonicStream** API to send streams of data to any number of interested applications, using SonicMQ® system as the transport mechanism, yet with no dependency on serializing the objects to a file on the sending side.

The Sonic Stream API uses the JMS Publish and Subscribe messaging model to break a byte stream of indeterminate length into a series of JMS BytesMessages. The message size limit is set by the stream application.

As illustrated in Figure 72, the sending application instance, the **Stream Publisher**, creates an output stream instance, and then writes data into that stream. The Sonic Stream API, in concert with the SonicMQ® Client Runtime, uses a topic as the destination on the **SonicMQ® Broker** for its stream. Receiving application instances, the **Stream Subscriber(s)**, create input stream instances and use the SonicMQ Client Runtime to subscribe to that topic, and then read data into their input stream instances. The StreamSubscriber application handles the data received to complete the data transfer.

Figure 72: Sonic Publisher Sending a Stream to Multiple Stream Subscribers to a Stream Topic



The SonicStream API is a lean process. Any requirements for handling interruption of a stream on the receiver side must be accomplished by the user's application code.

The size of the streamed data does need to be known as long as the sender application moderates the flow of data into the stream and the receiver applications offload the received streamed data out of their respective receiver applications at a rate that allows the receiver applications to stay within their allotted memory limits

A notification topic can be established so that a StreamSubscriber can post alerts of error conditions. The StreamPublisher's listener can read and act on the information. See [NotificationTopic](#) on page 327 for details.

Note: Scope of Usage of Sonic Streams: The delivery mode is limited to non-persistent. The subscriptions created by StreamSubscribers must be nondurable. Sessions created by StreamPublishers and StreamSubscribers must be nontransacted. Sonic® Stream functionality is only available for the SonicMQ® Java client. The advanced messaging feature of multitopic publish and multitopic subscribe are not supported for SonicStream applications. No support for recovery is provided to receivers

Common SonicStreamFactory Semantics

The following are the constructor signatures and the methods used by a Stream Publisher and Stream Subscribers.

Constructors

The constructors for `com.sonicsw.stream.SonicStreamFactory` let you choose to set the **StreamTopic** and the **ApplicationName** in the constructor:

- `SonicStreamFactory()`
- `SonicStreamFactory(String streamTopic)`
- `SonicStreamFactory(String streamTopic, String appName)`

Methods

The `setStreamTopic` method can be used to set or reset the stream topic.

StreamTopic

- `setStreamTopic(String topic)`
- `String getStreamTopic()`

`useTempStreamTopic`

- `useTempStreamTopic(boolean value)`

Instead of specifying the **StreamTopic**, setting this method to the value **true** specifies that a unique temporary topic be used to send and receive the SonicStream messages.

ApplicationName

- `setApplicationName(String appName)`
- `String getApplicationName()`

An application name can be added as a property to associated factory instances and stream controllers. The default **APPLICATION_NAME_PROPERTY** is **StreamsApp**.

NotificationTopic

- `setNotificationTopic(java.lang.String topic)`
- `String getNotificationTopic()`

Method to set the topic to which all notification messages are sent and received by stream controllers created through the factory instance. See [Notifications](#) on page 332 for more information.

SonicStream Interface

A **SonicStream** is a header message and a payload. The interface **SonicStream** lets you get information:

- `getHeaderMessage()` — Gets the JMS Message that is sent as the first message of the stream.
- `getStreamId()` — Gets the identifier that is used to distinguish a SonicStream instance (and its associated segments) from other SonicStream instances.

- **getInputStream()** — Gets the **java.io.InputStream** instance of this SonicStream instance.
- **getOutputStream()** — Gets the **java.io.OutputStream** instance of this SonicStream instance.
- **getNotificationDestination()** — Gets the JMS Destination used by this SonicStream instance. Data written to a SonicStream instance is published in the form of JMS TextMessages to this Destination. Likewise, data will be received, in the form of JMS TextMessages, from this Destination by a SonicStream instance.
- **getStreamStatus()** — Gets the current status of a SonicStream instance. The status is returned as a StreamStatus object.

Stream Publisher Semantics

The sender application uses a factory method to create an output stream controller instance. The topic used for publishing by the output stream instance can be passed in to the factory method. The sender application is responsible for creating a JMS Connection object, which is passed as an argument to the factory method. The session, producer, and consumer (for handling notifications from receivers) objects are created by the stream controller object.

The application gets an output stream instance from the controller, and then writes data to the output stream instance. The application can choose to set and get properties on the JMS message that is the stream header. The application can choose to monitor the progress of the data transfer by querying a stream status object obtained from the controller instance.

When the stream instance is complete, the application closes the stream. Once the stream is closed, the application can use the stream controller to create another stream.

When the application is complete, closing the stream controller closes the underlying JMS session, producer and consumer. The JMS Connection object used by the stream is closed by the sender application.

SonicStreamFactory

- **createSonicOutputStreamController(Connection con)**
- **createSonicOutputStreamController(Connection con, Destination streamTopic, Destination notifyTopic)**

Creates an instance of a **SonicOutputStreamController** object that is used to create stream instances.

SegmentSize

- **setSegmentSize(int segmentSize)**
- **int getSegmentSize()**

The **SonicOutputStream** supports the sending of data that is much larger than Sonic JMS message size limits because the stream is broken down into a series of segments (sometimes referred to as chunks.)

The segment size should be tuned to the buffer capacities on the broker. Set the number of bytes in a segment as a positive integer value to a maximum of 10 MB, the maximum supported message size for SonicMQ. The default segment size is 16384 bytes.

DeliveryMode

- **setDeliveryMode(int mode)**
- **int getDeliveryMode()**

On a sender, the delivery mode specifies the delivery mode of the **BytesMessages** that carry the message segments.

The delivery mode is, by default, **javax.jms.DeliveryMode.NON_PERSISTENT**. If you are using a security-enabled broker, the default is interpreted as **NON_PERSISTENT_SYNC**.

If you are using a broker that is not security-enabled, **NON_PERSISTENT** is interpreted as **NON_PERSISTENT_ASYNC**, meaning that no acknowledgement is expected by the publisher. You can set the **DeliveryMode** to **progress.message.jclient.DeliveryMode.NON_PERSISTENT_SYNC** so that the publisher method blocks to await acknowledgement by the broker.

The constant is typically used instead of the integer value:

- **setDeliveryMode(DeliveryMode.NON_PERSISTENT_SYNC)**

SonicOutputStreamController Interface

Methods in the **SonicOutputStreamController** interface include:

- **createStream(String id)** — Creates a **SonicOutputStream** instance with an identifier that will be in the header of every message sent through the stream.
- **registerNotificationListener(SonicStreamListener listener)** — Registers a listener that will receive asynchronous notifications. Only a single notification listener may be registered at a given time.
- **registerExceptionListener(SonicStreamExceptionListener listener)** — Registers a listener that will receive notification of asynchronous exceptions. Only a single exception listener may be registered at a given time.

The sending application should register one of each type of listener object with the output stream controller so that it can be notified of any asynchronous notifications or exceptions. The listener object handles notifications sent from any receiver applications, such as notice of a corrupted stream or a dropped connection. The sending application is responsible for handling such errors. For example, it might be appropriate for the sending application to clear the stream, and then write the original data to the output stream again.

The sender uses the **onStreamNotification(streamId, msg)** method of the **SonicStreamListener** interface to handle notification messages.

- **SonicStream createStream(String id)** — Creates a stream with the specified identifier
- **releaseStream(SonicStream ss)** — Closes the specified Stream instance
- **close()** — Closes the stream managed by the controller instance, JMS objects managed by the instance, and the controller itself

StreamStatus Interface

```
int getCurrentStreamStatus()
```

The **StreamStatus** interface provides methods to determine whether the stream has been created, the transfer is in progress, the transfer completed, and if a stream has encountered an error.

The **StreamStatus** interface also provides methods that get stream information:

- **String getStreamId()** — Identifies the stream instance.
- **long getTransferStart()** — Time when the transfer started.
- **long getSegmentDiscarded()** — Segments discarded by the stream.
- **long getBytesTransferred()** — Bytes transferred in the stream.
- **long getSegmentsTransferred()** — Segments transferred in the stream.
- **long getTransferEnd()** — Time when the transfer ended.
- **long getTransferTime()** — Elapsed time of the transfer.

Stream Subscriber Semantics

The receiver application uses a factory method to create an input stream controller instance, and is responsible for creating a JMS Connection object, passed in as an argument to the factory method. The topic used by the output stream instance can be passed to the factory method. The session, producer (for sending notifications to the sender's listener), and consumer objects are created by the stream controller object.

The application gets an instance of an input stream from the controller, and reads data from it. It can choose to access and read properties from the JMS message used as the stream header. The application can also choose to monitor the progress of the data transfer by querying a stream status object. When the application completes the reading of the stream (that is, when a read call returns **endOfStream**), it can get another input stream from the controller, or the application can close the stream.

When the application is complete, closing the stream controller closes the underlying JMS session, producer and consumer. The JMS **Connection** object used by the stream is the responsibility of the receiver application.

The receiving application instance does not have to re-assemble the segments from the sender, as that is handled by the **SonicInputStream** implementation. The receiving application receives the data in the same form as when it was written to the sending application's **SonicOutputStream** instance. (except in the event of a loss of connection or broker anomaly.)

SonicStreamFactory

- **createSonicInputStreamController(Connection con)**
- **createSonicInputStreamController(Connection con, Destination streamTopic, Destination notifyTopic)**

Creates an instance of a **SonicInputStreamController** object that is then used to obtain SonicInputStreams and to send notifications to the sender.

setDeliveryMode

- **setDeliveryMode(int mode)**
- **int getDeliveryMode()**

The delivery mode for the stream controller instance. On a stream receiver, the delivery mode applies to the notifications it emits.

setReadAheadWindowSize

- **setReadaheadWindowSize(int kbytes)**
- **int getReadaheadWindowSize()**

The **ReadAheadWindow** is a buffer on the receiver that helps to control memory consumption by the input stream controller. The content of the buffer consists of segments that have been read but not yet consumed by the application.

Set the number of kilobytes in the read ahead buffer as a positive integer value. The default read ahead window size is **500** kilobytes.

setSegment Timeout

- **setSegmentTimeout(int secs)**
- **int getSegmentTimeout()**

Sets how many seconds the **SonicInputStream** will wait for a segment of an opened stream to arrive. When the timeout expires, an exception is thrown by the **SonicInputStream**. The default value is **60** seconds.

Stream Handlers

- **getNextStream(long timeToWait)** — Returns the next available stream to be read.
- **getNextFullStream(long timeToWait)** — Returns the next available full stream to be read. Note that use of this method will likely result in the use of more heap memory than the **getNextStream** method, as the entire content of the **SonicStream** is assembled in memory before the **SonicStream** may be read. An exception will be thrown if the **getNextFullStream** method is invoked and the streams runtime detects that there is not enough space to hold the stream. Applications should ensure that enough space will be available by increasing the readahead window size via the **setReadaheadWindowSize** method of **SonicStreamFactory** prior to creating a **SonicInputController** instance.
- **isFullStreamAvailable()** — Boolean that indicates whether the entire content of a **SonicStream** is available to be read.
- **isPartStreamAvailable()** — Boolean that indicates whether at least a portion of a **SonicStream** is available to be read.
- **releaseStream(SonicStream ss)** — Closes the specified **SonicStream** instance and releases its associated resources.
- **close()** — Closes all streams managed by the **SonicInputStreamController** instance and closes the JMS objects created by the controller instance.

Notifications

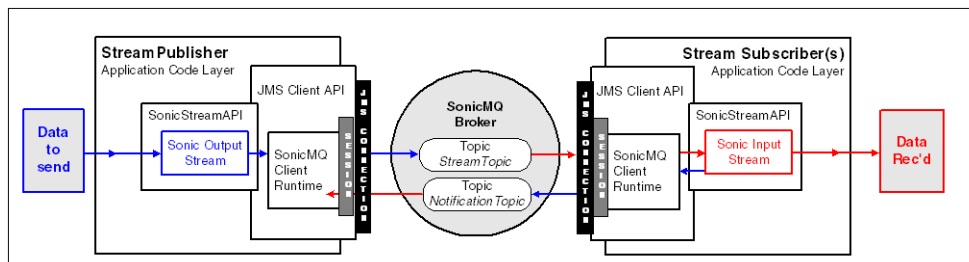
- `registerExceptionListener(SonicStreamExceptionListener listener)`
- `buildSenderNotification(int type, SonicStreamException ex)`
- `buildSenderNotification(int type, SonicStream ss, SonicStreamException ex)`

In the event of a corrupted stream or a dropped connection, the receiver application can send a notification to the sender application, as illustrated in Figure 73. The receiving application can register a listener object with the input stream controller. The listener object implementation must determine how to respond to the error condition.

A typical scenario is the listener object sends a notification from the input stream back to the sending application's stream object, indicating conditions such as an apparently corrupt stream or an interrupted connection. If the sender application is designed to resend the stream, the receiving application would clear the input stream by either closing the current stream (and wait for the next stream), or close the controller (and then create a new controller for a different stream topic, if appropriate).

If connection drops on the receiving application side, the receiving application is responsible for restoring the connection. (If the loss of connection is intermittent, the stream would realize missed segments which the broker would redeliver.) Once the connection is restored, the input stream implementation is responsible for sending a notification to the sending application. This notification is received by the listener registered with the sending application's output stream instance. It is up to the sending application how to proceed upon receipt of such a notification. The sending application might choose to write the original data to the stream again, or it might write it so that the data will only be sent to the receiving application instance(s) that did not get it.

Figure 73: StreamSubscriber Sending Notifications Back to the StreamPublisher



Managing Flow Control

A Stream Publisher might be flow-controlled at some point. While the expectation is that flow-controlled situations are typically transient, some may be of a longer duration due to bad connections or a slow network. To deal with such situations, an application can be configured so that subscribers can override the **FlowToDisk** setting of the broker through a setting on the **ConnectionFactory** used by the receiving application instance.

Handling Errors

The design of **SonicStreams** takes into account that the application designers typically know how they want to handle stream disruption. For example, if a receiver-side application suffers a dropped connection and subsequently reconnects to the Broker, then the receiver-side application instance would typically alert the sender-side of the disruption that took place. And the sender-side application instance could then resend the original data to the affected receiver-side application instance. Thus, the application could avoid resending the data to all receivers; the sending side would resend it only to the receiver application that did not receive the original data.

The JMS **Connection** supplied to the **SonicStreamFactory** should set the ping interval.

SonicStreams do not permit an application that uses multiple threads to access the same stream instance. Only one thread should write to a single output stream instance, and only one thread should read from a single input stream instance.

Samples of SonicStreams

Sonic provides sample applications that demonstrate **SonicStream** functionality that allows a publisher to send large objects to multiple subscribers using SonicMQ messaging. The Stream Publisher application supplies data to the SonicStreams runtime using the **java.io.OutputStream** API while the Stream Subscriber applications receive the data using the **java.io.InputStream** API.

There are two pairs of sample applications for the **SonicStream** API:

- **SonicStreams** — The **StreamSender** and **StreamReceiver** applications demonstrate basic functionality where the sender sends a hashed memory object and the receiver receives it.
- **SonicStreams with Retry** — The **StreamSender_Retry** and **StreamReceiver_Retry** applications demonstrates advanced functionality. The sender sends streams, and accepts requests for resend from stream receivers. Resending provides an example of the handling of errors in stream transfers.

The sample applications are provided in source code text form so that you can use them as a basis for your **SonicStreams** applications. The topic name **StreamTopic** is arbitrary. The source code is extensively annotated with comments to help clarify their patterns and behaviors.

Note: User authentication on security-enabled brokers — The connection to the broker you use for these samples requires valid user credentials if you chose to enable security when installing or configuring the broker. If security is enabled, you can either setup the sample user identities specified in the command line for each sample, or supply your own user identities. If you have not created or maintained users, the default user is **Administrator** with the password **Administrator**.

Note: Hostname and port — The samples refer to **localhost**. If the broker where you want to connect is on a host other than **localhost**, use that host's name. The default port defined at installation of the broker is **2506**. Use a port number that corresponds to an acceptor on the broker you want to use. If your acceptor definition is not [tcp://localhost:2506](http://localhost:2506), you need to specify the URL in a sample's command line with the **-b** parameter, such as: `..\..\SonicMQ StreamReceiver -u user -p pwd -st StreamTopic -b tcp://myHost:3456`

SonicStreams Sample

In this pair of sample applications, you are guided to start the receiver application and then start the sender application.

1. Start the SonicMQ broker you want to use in the sample.
2. Open a command line console window, change directory to to
`<install_dir>/MQ10.0/samples/TopicPubSub/SonicStreams.`
3. Enter:
 - On a Windows system: **..\SonicMQ StreamReceiver -u user -p pwd -st StreamTopic**
 - On a UNIX® or Linux® system: **../SonicMQ.sh StreamReceiver -u user -p pwd -st StreamTopic**
 - The **StreamReceiver's** console window displays:
 - **Waiting 120 sec. for a stream...**
4. Open another command line console window, change directory to the same path.
5. Enter:
 - On a Windows system: **..\SonicMQ StreamSender -u user -p pwd -st StreamTopic**
 - On a UNIX or Linux system: **../SonicMQ.sh StreamSender -u user -p pwd -st StreamTopic**
6. The **StreamSender's** console window displays:
 - a. **Generated test data**
 - b. **Sending an object**
 - c. **Sending object: MyObject_1nnn**
 - d. *****Notification: Receiver StreamsApp started receiving MyObject_1nnn**
 - e. **CompletedSend;**
 - f. **Stream MyObject_1nnn: bytesSent= 3189043 chunksSent= 392 time= 2173**
 - g. *****Notification: Receiver StreamsApp completed receiving MyObject_1nnn**
 - h. **Sleeping 5 sec**
7. The **StreamReceiver's** console window displays:
 - a. **Receiving Stream MyObject_1nnn**
 - b. **Received object of type java.util.Hashtable; Stream MyObject_1nnn: bytesReceived= 3189043 chunks= 392 time= 2233**
 - c. **Received HashTable; htSize= 100000**
 - d. **Waiting 120 sec. for a stream...**

8. The **StreamSender's** console window displays:
 - a. *****Notification: Receiver StreamsApp completed receiving MyObject_1nnn**
 - b. **Sleeping 5 sec**
 - c. Both applications wait for a few seconds.
9. The **StreamSender's** console window displays:
 - a. **Sending another object**
 - b. **Sending object: MyObject_2nnn**
 - c. *****Notification: Receiver StreamsApp started receiving MyObject_2nnn**
 - d. **CompletedSend;**
 - e. **Stream MyObject_2nnn: bytesSent= 7189043 chunksSent= 880 time= 3395**
10. The **StreamReceiver's** console window displays:
 - a. **Receiving Stream MyObject_2nnn**
 - b. **Received object of type java.util.Hashtable; Stream MyObject_2nnn: bytesReceived= 7189043 chunks= 879 time= 3395**
 - c. **Received HashTable; htSize= 200000**
11. The receiver application exits.
12. The **StreamSender's** console window displays:
 - a. **Sleeping 10 sec**
 - b. *****Notification: Receiver StreamsApp completed receiving MyObject_2nnn**
13. After the sleep period times out, the sender application exits.

You can extend this sample by:

- Starting more instances of the receiver application. On each of the receivers, add the **-n** parameter to provide a unique Application Name in the notifications displayed in the **StreamSender's** console window.
- Modify **StreamSender.java** to define different object sizes, or establish different data streams and modified handling of the received stream data. You can also adjust the segment size to determine ideal segments for your stream data. Then compile and run the changed files to evaluate your changes.

SonicStreams Sample With Retry

This pair of sample applications show how their retry logic handles the behaviors when you start the sender before the receiver, and how interruptions to the sender, broker, and receiver can be handled.

1. Start the SonicMQ broker you want to use in the sample.
2. Open a command line console window, change directory to `install_dir/MQ10.0/samples/TopicPubSub/SonicStreams`.

3. Enter:

- On a Windows system:
 - `..\..\SonicMQ StreamReceiver_Retry -u user -p pwd -st StreamTopic`
- On a UNIX® or Linux® system:
 - `../../SonicMQ.sh StreamReceiver_Retry -u user -p pwd -st StreamTopic`

4. Open another command line console window, change directory to the same path.

5. Enter:

- On a Windows system:
 - `..\..\SonicMQ StreamSender_Retry -u user -p pwd -st StreamTopic`
- On a UNIX® or Linux® system:
 - `../../SonicMQ.sh StreamSender_Retry -u user -p pwd -st StreamTopic`

The sender streams consist of a few different hypothesized `DataOutputStreams`. Note that there are occasional pauses in the sending and receiving of the streams to make the demonstration behaviors easier to observe.

Console Information in an Uninterrupted Transfer

Note: The following sender and receiver listing of console information result when the transfers complete successfully. Items in bold are user actions. Lines with just ... indicate that a section of the information was eliminated from this listing.

Sender

```
Start
StreamSender_Retry
Attempting to create connection...
Created
new controller; dest= StreamTopic
Sending stream:
Stream_10000;dest= StreamTopic
***Notification:
Receiver StreamsApp started receiving Stream_10000
Stream_10000:
Wrote 1000
Stream_10000: Wrote 2000
Stream_10000: Wrote
3000
...
Stream_10000: Wrote 8000
Stream_10000:
Wrote 9000
Stream_10000: Wrote 10000
CompletedSend;
Stream Stream_10000: bytesSent= 80004 chunksSent= 83 time= 10205
Sending
stream: Stream_20000;dest= StreamTopic
Stream_20000:
Wrote 1000
***Notification: Receiver StreamsApp
completed receiving Stream_10000
***Notification:
```



```

Receiver StreamsApp started receiving Stream_20000
Stream_20000:
Wrote 2000
Stream_20000: Wrote 3000
...
Stream_20000:
Wrote 18000
Stream_20000: Wrote 19000
Stream_20000:
Wrote 20000
CompletedSend; Stream Stream_20000:
bytesSent= 160004 chunksSent= 163 time= 20309
Sending
stream: Stream_40000;dest= StreamTopic
Stream_40000:
Wrote 1000
***Notification: Receiver StreamsApp
completed receiving Stream_20000
***Notification: Receiver
StreamsApp started receiving Stream_40000
Stream_40000:
Wrote 2000
Stream_40000: Wrote 3000
...
Stream_40000:
Wrote 38000
Stream_40000: Wrote 39000
Stream_40000:
Wrote 40000
CompletedSend; Stream Stream_40000:
bytesSent= 320004 chunksSent= 323 time= 40510
Waiting
30000 for stream requests...
***Notification: Receiver
StreamsApp completed receiving Stream_40000
Waiting
30000 for stream requests...
Waiting 30000 for stream
requests...
...
Waiting 30000 for
stream requests.....CTRL-C

```

Receiver

```

Start
StreamReceiver_Retry
Attempting to create connection...
Created
new controller; dest= StreamTopic
Sending retry
request for streamId ALL to TheNotifyTopic
Waiting
60 sec for more Streams on topic StreamTopic
Receiving
stream: Stream_10000
Stream_10000: Read 1000
Stream_10000:
Read 2000
Stream_10000: Read 3000
...
Stream_10000:
Read 8000
Stream_10000: Read 9000
Stream_10000:
Read 10000
*** Completed successful receive; Stream
Stream_10000: bytesReceived= 80004 chunksReceived= 83 time= 11196
Waiting
60 sec for more Streams on topic StreamTopic

```

```
Receiving
stream: Stream_20000
Stream_20000: Read 1000
Stream_20000:
Read 2000
Stream_20000: Read 3000
...
Stream_20000:
Read 18000
Stream_20000: Read 19000
Stream_20000:
Read 20000
*** Completed successful receive; Stream
Stream_20000: bytesReceived= 160004 chunksReceived= 163 time= 21301
Waiting
60 sec for more Streams on topic StreamTopic
Receiving
stream: Stream_40000
Stream_40000: Read 1000
Stream_40000:
Read 2000
Stream_40000: Read 3000
...
Stream_40000:
Read 38000
Stream_40000: Read 39000
Stream_40000:
Read 40000
*** Completed successful receive; Stream
Stream_40000: bytesReceived= 320004 chunksReceived= 323 time= 41521
Waiting
60 sec for more Streams on topic StreamTopic
No
more streams available; ...
Performing retries:
countToRetry= 0
Shutting down...
```

Experimenting with Interruptions

When the receiver starts up, it sends a notification to the sender requesting all streams. If the receiver detects an error, it will send a notification to the sender requesting a resend of the stream to a temporary topic. If the connection is dropped, the receiver application will reestablish the connection. The sender keeps a list of retry requests and services them one at a time.

The following listings show the information in the sender and receiver console windows when:

- The stream receiver is stopped and then restarted.
- The stream sender is stopped and then restarted.
- The broker with the streamTopic is stopped and then restarted.

Console Information in an Transfer Where the Receiver is Interrupted

Sender

```
Start
StreamSender_Retry
Attempting to create connection...
Created
new controller; dest= StreamTopic
Sending stream:
```

```

Stream_10000;dest= StreamTopic
***Notification:
Receiver StreamsApp started receiving Stream_10000
Stream_10000:
Wrote 1000
Stream_10000: Wrote 2000
Stream_10000: Wrote
3000
...
Stream_10000: Wrote 8000
Stream_10000:
Wrote 9000
Stream_10000: Wrote 10000
CompletedSend;
Stream Stream_10000: bytesSent= 80004 chunksSent= 83 time= 10195
Sending
stream: Stream_20000;dest= StreamTopic
Stream_20000:
Wrote 1000
***Notification: Receiver StreamsApp
retry request for streamId: ALL replyDest= StreamTopic
Stream_20000:
Wrote 2000
Stream_20000: Wrote 3000
...
Stream_20000:
Wrote 18000
Stream_20000: Wrote 19000
Stream_20000: Wrote
20000
CompletedSend; Stream Stream_20000: bytesSent=
160004 chunksSent= 163 time= 20360
Sending stream:
Stream_40000;dest= StreamTopic
***Notification:
Receiver StreamsApp started receiving Stream_40000
Stream_40000:
Wrote 1000
Stream_40000: Wrote 2000
Stream_40000:
Wrote 3000
...
Stream_40000: Wrote 7000
Stream_40000:
Wrote 8000

```

Receiver

```

Start
StreamReceiver_Retry
Attempting to create connection...
Created
new controller; dest= StreamTopic
Sending retry
request for streamId ALL to TheNotifyTopic
Waiting
60 sec for more Streams on topic StreamTopic
Receiving
stream: Stream_10000
Stream_10000: Read 1000
Stream_10000:
Read 2000
Stream_10000: Read 3000
...
Stream_10000:
Read 8000
Stream_10000: Read 9000
Stream_10000:
Read 10000
*** Completed successful receive; Stream

```

```
Stream_10000: bytesReceived= 80004 chunksReceived= 83 time= 20200
Waiting
60 sec for more Streams on topic StreamTopic
Receiving
stream: Stream_20000
Stream_20000: Read 1000
Waiting
60 sec for more Streams on topic StreamTopic
*****
Releasing duplicate stream Stream_10000
Waiting
60 sec for more Streams on topic StreamTopic
onStreamException:
StreamId= Stream_10000; error: errorcode= 3
com.sonicsw.stream.SonicStreamException:
Received segment 1 for
stream Stream_10000 ; stream
does not exist; msg discarded
Receiving stream: Stream_20000
Stream_20000:
Read 1000
Stream_20000: Read 2000
Stream_20000:
Read 3000
...
Stream_20000: Read 8000
Stream_20000:
Read 9000
Stream_20000: Read 10000
CTRL-C
Restart
StreamReceiver_Retry
Attempting to create connection...
Created
new controller; dest= StreamTopic
Sending retry
request for streamId ALL to TheNotifyTopic
Waiting
60 sec for more Streams on topic StreamTopic
onStreamException:
StreamId= Stream_20000; error: errorcode= 3
com.sonicsw.stream.SonicStreamException:
Received segment 121 fo
r stream Stream_20000 ;
stream does not exist; msg discarded
Receiving stream: Stream_40000
Stream_40000:
Read 1000
Stream_40000: Read 2000
CTRL-C
```

Console Information in an Transfer Where the Sender is Interrupted

Sender

```
Start
StreamSender_Retry
Attempting to create connection...
Created
new controller; dest= StreamTopic
Sending stream:
Stream_10000; dest= StreamTopic
***Notification:
Receiver StreamsApp started receiving Stream_10000
Stream_10000:
Wrote 1000
```

```

Stream_10000: Wrote 2000
Stream_10000: Wrote
3000
...
Stream_10000: Wrote 7000
Stream_10000:
Wrote 8000
Stream_10000: Wrote 9000
Stream_10000:
Wrote 10000
CompletedSend; Stream Stream_10000:
bytesSent= 80004 chunksSent= 83 time= 19208
Sending
stream: Stream_20000;dest= StreamTopic
Stream_20000:
Wrote 1000
***Notification: Receiver StreamsApp
completed receiving Stream_10000
***Notification:
Receiver StreamsApp started receiving Stream_20000
Stream_20000:
Wrote 2000
CTRL-C
Restart StreamSender_Retry
Attempting
to create connection...
Created new controller;
dest= StreamTopic
Sending stream: Stream_10000;dest= StreamTopic
Stream_10000:
Wrote 1000
Stream_10000: Wrote 2000
Stream_10000:
Wrote 3000
...
Stream_10000: Wrote 8000
Stream_10000:
Wrote 9000
Stream_10000: Wrote 10000
CompletedSend;
Stream Stream_10000: bytesSent= 80004 chunksSent= 83 time= 10135
Sending
stream: Stream_20000;dest= StreamTopic
***Notification:
Receiver StreamsApp started receiving Stream_20000
Stream_20000:
Wrote 1000
Stream_20000: Wrote 2000
Stream_20000:
Wrote 3000
...
Stream_20000: Wrote 13000
Stream_20000:
Wrote 14000
Stream_20000: Wrote 15000
***Notification:
Receiver StreamsApp retry request for streamId: ALL replyDest= StreamTopic
Stream_20000:
Wrote 16000
Stream_20000: Wrote 17000
Stream_20000:
Wrote 18000
Stream_20000: Wrote 19000
Stream_20000:
Wrote 20000
CompletedSend; Stream Stream_20000:
bytesSent= 160004 chunksSent= 163 time= 29943
Sending
stream: Stream_40000;dest= StreamTopic
Stream_40000:
Wrote 1000

```

```
***Notification: Receiver StreamsApp
started receiving Stream_40000
Stream_40000: Wrote 2000
Stream_40000:
Wrote 3000
CTRL-C
```

Receiver

```
Start
StreamReceiver_Retry
Attempting to create connection...
Created
new controller; dest= StreamTopic
Sending stream:
Stream_10000;dest= StreamTopic
***Notification:
Receiver StreamsApp started receiving Stream_10000
Stream_10000:
Wrote 1000
Stream_10000: Wrote 2000
Stream_10000:
Wrote 3000
...
Stream_10000: Wrote 8000
Stream_10000:
Wrote 9000
Stream_10000: Wrote 10000
CompletedSend;
Stream Stream_10000: bytesSent= 80004 chunksSent= 83 time= 10195
Sending
stream: Stream_20000;dest= StreamTopic
Stream_20000:
Wrote 1000
***Notification: Receiver StreamsApp
retry request for streamId: ALL replyDest= StreamTopic
Stream_20000:
Wrote 2000
Stream_20000: Wrote 3000
Stream_20000:
Wrote 4000
...
Stream_20000: Wrote 18000
Stream_20000:
Wrote 19000
Stream_20000: Wrote 20000
CompletedSend;
Stream Stream_20000: bytesSent= 160004 chunksSent= 163 time= 20360
Sending
stream: Stream_40000;dest= StreamTopic
***Notification:
Receiver StreamsApp started receiving Stream_40000
Stream_40000:
Wrote 1000
Stream_40000: Wrote 2000
Stream_40000:
Wrote 3000
...
Stream_40000: Wrote 7000
Stream_40000:
Wrote 8000
```

Console Information in an Transfer Where the Broker is Interrupted

Sender

```

Start
StreamSender_Retry
Attempting to create connection...
Created
new controller; dest= StreamTopic
Sending stream:
Stream_10000;dest= StreamTopic
Stream_10000: Wrote 1000
Stream_10000:
Wrote 2000
Stream_10000: Wrote 3000
Stream_10000:
Wrote 4000
***Notification: Receiver StreamsApp
retry request for streamId: ALL replyDest= StreamTopic
Stream_10000:
Wrote 5000
Stream_10000: Wrote 6000
Stream_10000: Wrote
7000
Stream_10000: Wrote 8000
Stream_10000:
Wrote 9000
Stream_10000: Wrote 10000
CompletedSend;
Stream Stream_10000: bytesSent= 80004 chunksSent= 83 time= 10225
Sending
stream: Stream_20000;dest= StreamTopic
***Notification:
Receiver StreamsApp started receiving Stream_20000
Stream_20000:
Wrote 1000
Stream_20000: Wrote 2000
Stream_20000: Wrote
3000
onStreamException: StreamId= Stream_20000;
error: errorcode= 7 com.sonicsw.stream.SonicStreamException: JMSEException
Attempting
to create connection...
Cannot connect to broker:
localhost:2506. Pausing 10 seconds before retry.
Attempting
to create connection...
Cannot connect to broker:
localhost:2506. Pausing 10 seconds before retry.
...
Cannot
connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting
to create connection...
Created new controller;
dest= StreamTopic
Sending stream: Stream_40000;dest= StreamTopic
Stream_40000:
Wrote 1000
***Notification: Receiver StreamsApp
started receiving Stream_40000
Stream_40000: Wrote 2000
Stream_40000:
Wrote 3000
...
Stream_40000: Wrote 8000

```

```
Stream_40000:
Wrote 9000
CTRL-C
```

Receiver

```
Start
StreamReceiver_Retry
Attempting to create connection...
Created
new controller; dest= StreamTopic
Sending retry
request for streamId ALL to TheNotifyTopic
Waiting
60 sec for more Streams on topic StreamTopic
onStreamException:
StreamId= Stream_10000; error: errorcode= 3
com.sonicsw.stream.SonicStreamException:
Received segment 33 for
  stream Stream_10000 ;
stream does not exist; msg discarded
Receiving stream: Stream_20000
Stream_20000:
Read 1000
Stream_20000: Read 2000
onStreamException:
StreamId= Stream_20000; error: errorcode= 7
com.sonicsw.stream.SonicStreamException:
JMSEException
Attempting to create connection...
Cannot
connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting
to create connection...
...
Cannot
connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting
to create connection...
Created new controller;
dest= StreamTopic
Sending retry request for streamId
ALL to TheNotifyTopic
Waiting 60 sec for more Streams
on topic StreamTopic
Receiving stream: Stream_40000
Stream_40000: Read
1000
Stream_40000: Read 2000
Stream_40000:
Read 3000
...
Stream_40000: Read 8000
CTRL-C
```


Hierarchical Name Spaces

This chapter provides information about hierarchical name spaces, which allow you to create a hierarchy of contents by delimiting nodes when you name a topic. This feature can be useful in the naming and management of topics.

For details, see the following topics:

- [About Hierarchical Name Spaces](#)
- [Publishing a Message to a Topic](#)
- [Broker Management of Topic Hierarchies](#)
- [Subscribing to Nodes in the Topic Hierarchy](#)
- [Examples of a Topic Name Space](#)

About Hierarchical Name Spaces

Hierarchical name spaces are a topic-grouping mechanism available with SonicMQ® system. When you use topics in the Pub/Sub domain, the publisher, broker, and subscriber all adhere to the JMS standards. But SonicMQ® system 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.

Advantages of Hierarchical Name Spaces

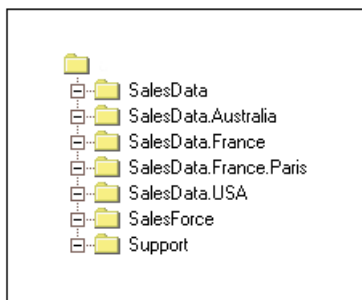
Naming conventions become cumbersome to work with when long strings are passed around as identifiers. SonicMQ offers the ability to use a naming and directory service with the naming and management of topics. As a result, topics are easier to specify and control for clients and are correspondingly faster 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 *Aurea 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 the following figure.

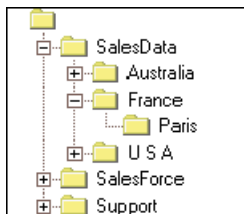
Figure 74: Topic Structure Without Hierarchies



Without hierarchies, many topics are stacked onto one level. When many topics are used, it gets 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 the following figure.

Figure 75: 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 a Message to a Topic

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 use of hierarchy delimiters and deprecates the use of a few special characters and topic names.

Topic Notation that Enables Topic Hierarchies

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 *Aurea 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.

For example, the **Chat** sample uses the topic name **jms.samples.chat**.

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 to the hierarchy (period-separated 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 unwieldy 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 *Aurea SonicMQ Configuration and Management Guide*.

Topic Syntax and Semantics

The following naming conventions apply to topic naming:

- Case sensitive — Topic names are case sensitive (like the Java language). 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

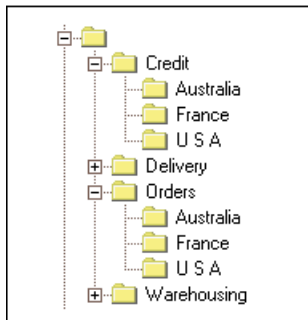
Topic hierarchies empower the broker in two significant ways:

- 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 that would otherwise be 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 JMS standard way with the **Topic** and the **TopicSubscriber** methods. As shown in the following figure, to get messages published for **U S A Credit**, use the topic name **Credit.U S A**.

Figure 76: 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).

The intent of the template characters is to allow a set of managed topics to exist in a message system in a way that lets subscribers choose broad subscription parameters that will 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 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.

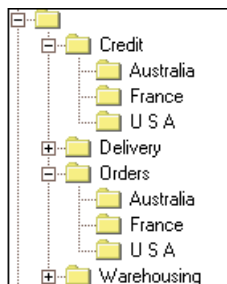
SonicMQ will deliver a message to more than one message handler if the message's topic matches bindings from multiple handlers.

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 the following figure 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 77: Symmetric Topic Structure



Template Character for All Topics at a Content Level

Using the strict topic hierarchy shown in the above figure, 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 will deliver messages sent to these destinations:

- **Credit.Australia**
- **Credit.France**
- **Credit.U S A**

Template Character for a Topic at a Content Level

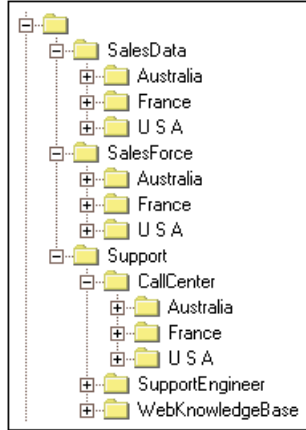
A subscription to the topic expression ***.U S A** in the hierarchy in the above figure selects all **U S A** topics at the second level of the hierarchy. This subscription will deliver messages sent to these destinations:

- **Credit.U S A**
- **Orders.U S A**

Using Template Characters in Asymmetric Topic Hierarchies

When there are several topic levels, as shown in the following figure, subscribing to all the **U S A** topics is complicated by an inconsistent topic-naming structure.

Figure 78: 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, such 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 would 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 **#** will receive 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 that are published on the broker host by subscribing to **jms.samples.#**. The sample does not subscribe to **#**, because such a subscription would include 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 certain, 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. You can:

- 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.)

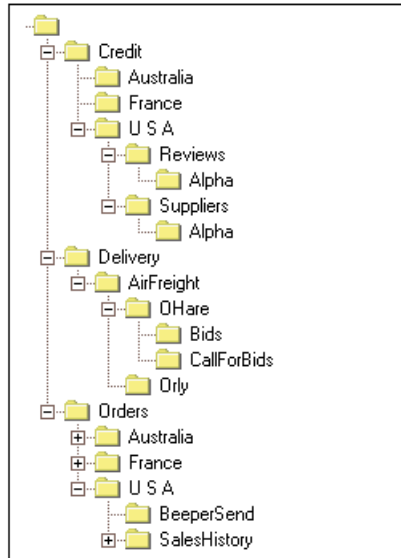
Examples of multiple template characters in an expression are:

- 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 a Topic Name Space

The hypothetical topic hierarchy shown in the following figure has nodes that might represent levels of responsibility in the enterprise.

Figure 79: A Sample Hierarchy of Topics



Publishing Messages to a Hierarchical Topic

The publisher produces messages to a single fully qualified topic, such as:

```
static final String MESSAGE_TOPIC = "Credit.U S A.Customers";
```

Business cases where a publisher might use a hierarchical topic 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 SA.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 final String MESSAGE_TOPIC = "Credit.U S A";
```

or use template characters to subscribe to sets of topics:

```
private static final String MESSAGE_TOPIC = "Credit.*";
```

Business cases where a subscriber gains 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.

Distributed Transactions Using XA Resources

This chapter provides information about using distributed transactions and the **XAResource** class. For details, see the following topics:

- [About Distributed Transactions](#)
- [Interfaces for Distributed Transactions](#)
- [In-doubt Global Transactions](#)
- [Distributed Transactions Models](#)
- [Running the Distributed Transaction Sample](#)

About Distributed Transactions

Distributed transaction processing (DTP) allows a set of messages from heterogeneous sessions to form a composite transaction.

An example of a distributed transaction is a transfer of funds between bank accounts. Withdrawal from one account and deposit into another account comprise one balanced transaction. Every effort must be taken to assure that the transaction—even though it might involve two different banks can—and likely will—complete successfully. If it cannot succeed, no part of it can be recorded.

General Properties of a Transaction

A distributed transaction is characterized by the same properties as a simple transaction (a series of messages sent and/or received on one session thread):

- **Atomic** — Either all of the work in the transaction must complete, or none of the work must complete.
- **Consistent** — The transaction ensures that all participants in the transaction are in the same state.
- **Isolated** — A transaction executes in isolation and does not affect other concurrent transactions.
- **Durable** — A transaction will not be lost subsequent to a system failure.

These properties are often referred to as **ACID** properties.

Transaction Types

A single SonicMQ application can contain both local and global transactions.

Local Transaction

A local transaction involves a single resource manager. In SonicMQ, a transaction performed in a session with the transacted parameter set to true is a local transaction.

Global Transaction

A global transaction involves dispersed resources in the transaction. It is often referred to as a distributed transaction. A distributed transaction system typically relies on an external transaction manager to coordinate the participants in a transaction.

Components of Distributed Transactions

Distributed transactions have as many as five components. Each contributes to the distributed transaction processing system by implementing different sets of transaction APIs and functionalities. The Java Transaction Service (JTS) describes these components as follows:

- A **transaction manager** provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.
- An **application server** (or TP monitor) provides the infrastructure that supports the application run-time environment including transaction state management. An example of such an application server is an EJB server.
- A **resource manager (RM)** is an entity that manages data or some other kind of resource through a resource adapter. SonicMQ is a resource manager, implementing a transaction resource interface—the **XAResource** interface of the Java Transaction API (JTA)—that the transaction manager uses to communicate transaction association, transaction completion, and recovery work.
- A transactional **application** in an application server environment relies on the application server to provide transaction management support through transaction attribute settings—for example, an application developed using the Enterprise JavaBeans (EJB) component architecture. In

addition, other standalone Java client programs might want to control their transaction boundaries using a high-level interface provided by the application server or the transaction manager.

- A **communication resource manager** (CRM) supports transaction context propagation and access to the transaction service for incoming and outgoing requests.

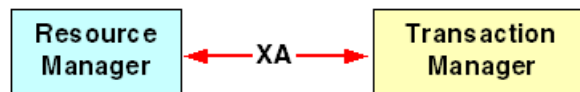
From the transaction manager's perspective, the actual implementation of the transaction services does not need to be exposed. Only high-level interfaces need to be defined to allow transaction demarcation, resource enlistment, synchronization, and recovery process to be driven by the users of the transaction services.

Using XA Resources

A SonicMQ application has access to all the required components for distributed transactions. When a SonicMQ application imports `javax.transaction.xa`, the `XAResource` and `XID` classes are included so that—together with the XA connection factories, connections, and sessions in `javax.jms`—the Java Transaction API is enabled.

A SonicMQ application initiates a transaction by using the JTA to communicate with the transaction manager. The RM uses the XA protocol to connect to the transaction manager (TM), as shown in the following figure.

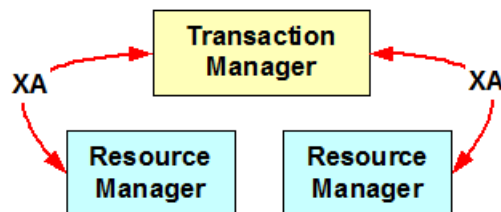
Figure 80: Resource Manager Connected to a Transaction Manager



As RMs on diverse threads will participate in the distributed transaction, it is crucial that the TM establish and maintain the state of the transaction as it evolves. A **transaction context** logically envelopes all the operations performed on transactional resources during the transaction.

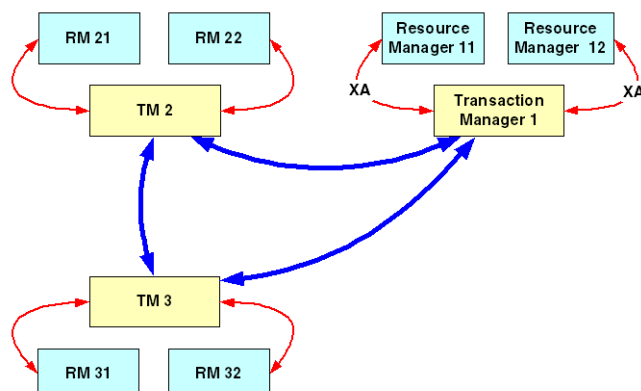
An application server can bring additional resources into the transaction, referred to as **resource enlistment**. In the following figure, another RM is participating in `xid1` in an XA connection to the same TM as the owner.

Figure 81: Enlistment of a Second RM to a Distributed Transaction



The transaction context and resource enlistment can extend to other transaction managers, as shown in the following figure.

Figure 82: Distributed Transaction with Multiple Transaction Managers



Messages sent in the transaction context are not permanently recorded. Messages received in the transaction context are not acknowledged until the owner ends the transaction demarcation and passes control to its TM. This TM can coordinate with the other involved TMs and Resource Managers to prepare and commit the global transaction.

The transaction owner explicitly ends the global transaction. If the transaction involved a single transaction manager, the act of telling the TM to commit or rollback the transaction would handle the transaction state on all the participating branches.

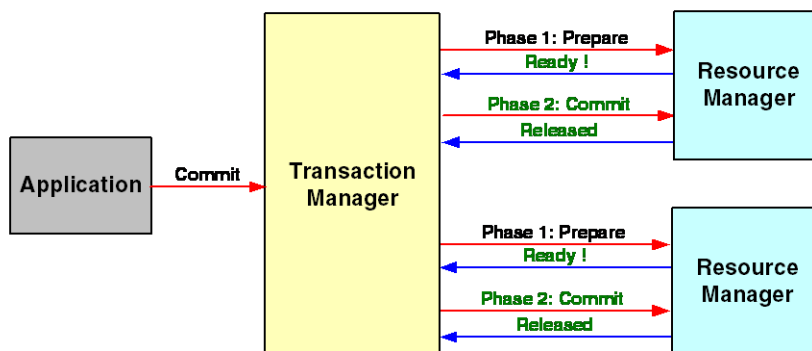
A global transaction that involves multiple resource managers needs a two-phase commit protocol to request that the participants prepare for completion, and, when all are prepared, perform the commitment. It is possible that a failure during the commitment process leaves the state of the global transaction indoubt.

Two Phase Commit

When the transaction manager prepares the transaction, each participating resource manager is polled to see if commitment is likely to succeed.

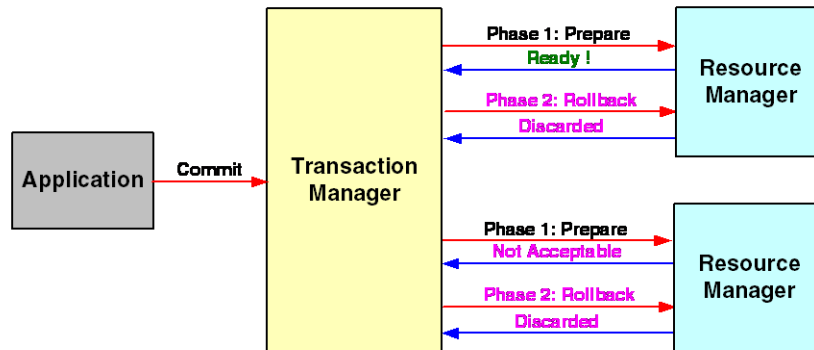
In the following figure, both resource managers, when commanded to prepare their contribution to transaction **xid1**, responded positively. The global commit for **xid1** by the owner's transaction manager commits each branch of the global transaction.

Figure 83: Two-phase Commit That Completed Successfully



In the following figure, one of the resource managers indicated that its portion of the global transaction could not be completed. As a result, a global rollback is issued for each branch of xid1.

Figure 84: Two-phase Commit That Completed Unsuccessfully



Interfaces for Distributed Transactions

SonicMQ supports the JTA **XAResource** interface and JMS XA SPI.

javax.transaction.xa Interfaces

The Java Transaction API (JTA) mapping of the industry standard XA interface based on the X/Open CAE Specification provides, among other features:

- An external transaction manager and sophisticated application capability to demarcate global transactions
- Assignment of a session to a distributed transaction
- Preparation and commitment of one or more transactions
- Recovery of a transaction in progress

The interfaces in **javax.transaction.xa** are **XAResource**, **Xid**, and **XAException**. See Sun's online JavaDoc at <http://java.sun.com/products/jta/javadocs-1.0.1/index.html> for the exposed fields, methods and constructors in these interfaces.

JMS XA SPI Interface

The JMS service provider interface for XA exposes **XAResource** to application servers and expert JMS clients (through its **XAConnectionFactory**, **XAConnection** and **XASession**) thereby providing access to the regular JMS counterpart (such as a **Session** object) and coordinating the regular session and **XAResource** through **XASession**.

The hierarchy of the JMS XA interfaces, as shown in the following figure, is similar to non-XA connections and sessions.

Figure 85: XASessions in XAConnections from XAConnectionFactory



XAConnectionFactory

XASonicMQ exposes its JTS support through the JMS **XAConnectionFactory** which distributed transactions—and possibly application servers—use to create **XASessions**.

An **XAConnectionFactory** object is an administered object similar to a **ConnectionFactory**. See [Connection Factories and Connections](#) for information about lookup as a serialized object in a file store or on a JNDI LDAP server.

XAConnection

An **XAConnection** is similar to a **Connection** except that is by definition transacted. You can choose a default or a specified user identity.

Base the ConnectID for an XAConnection on Its Connection

- An **XAConnection**'s connectID is the **connectID** of the **Connection**, prefixed by "XAC_". In this code sample, **c**'s connectID is "myConID" and **xac**'s connectID is "XAC_myConID":

```

XAConnectionFactory xacf =
    new progress.message.jclient.xa.XAConnectionFactory (url, "myConID",
    ... ) ;

XAConnection xac = xacf.createXAConnection();
Connection c = xac;
  
```

Load Balancing Is Inactive for an XA Client

Load balancing offers the ability to connect to another broker when a broker is overloaded. But the nature of a global transaction is that the originating thread and the **xid** are mapped to a specific broker. To accommodate global transactions, load-balancing is turned off in the client by default. If an XA client turns on load-balancing, a **JMSEException** is thrown when you create XA connections.

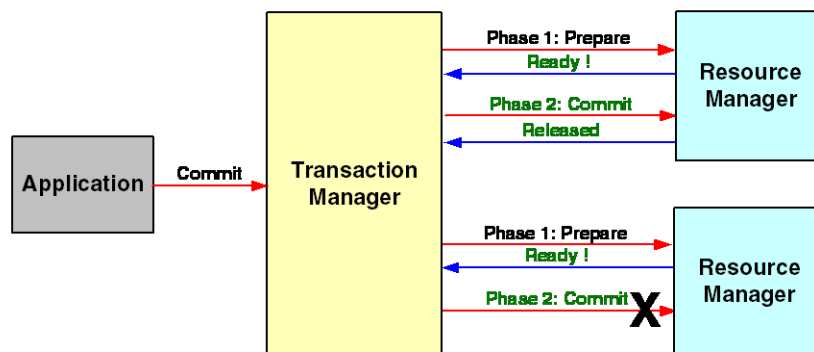
XASession

The **XASession** interface provides access to the **Session** object and a **javax.transaction.xa.XAResource** object which controls the session's transaction context. A client uses the JMS **Session** obtained from the **XASession** to perform its JMS work. The **XAResource** is used to assign the session to a distributed transaction, and to prepare and commit (or roll back) work on the transaction transparently. What appears as a regular JMS Session is actually controlled by the transaction management of the underlying **XASession**.

In-doubt Global Transactions

If a global transaction succeeded in phase one of the two-phase commit, the physical connections could fail such that part of the global transaction is recorded and part of it is indeterminate. The commitment might have been executed but failed before notifying the TM. The following table illustrates an in-doubt transaction that occurs during a two-phase commit.

Figure 86: In-doubt Transaction During Two-Phase Commit



In this case, one branch of the global transaction might have been committed while the other branch did not.

SonicMQ Can Complete In-doubt Transaction Branches

SonicMQ provides administrative capabilities to members of the **TxnAdministrator** group to review and manipulate the uncommitted branches. For more information about transaction administrators and resolution of indoubt transaction branches in the *Aurea SonicMQ Configuration and Management Guide*.

Access Control Group for Transaction Administrators

The SonicMQ administrative tools can assign users to a special purpose user group, **TxnAdministrators**, to view and handle indoubt transactions. See the *Aurea SonicMQ Configuration and Management Guide* for information about the transaction administrators.

Transaction Recovery

If the transaction owner becomes unable to continue, a transaction manager can invoke a **recover** method to a resource manager to determine the identity (**xidnnn**) and status of transaction branches.

The **TMSTARTRSCAN**, **TMENDRSCAN** and **TMNOFLAGS** flags are supported in the implementation of the **XAResource.recover(int flag)** method in compliance with the XA specification. A given recovery scan must be made by the same **XAResource** instance.

The following examples illustrate how these flags return results from the recover method:

Example 1: TMNOFLAGS

When **recover(TMNOFLAGS)** is called without having called **recover(TMSTARTRSCAN)**, all in-doubt transaction Xids are retrieved. This is the preferred technique for recovery.

The recovery scan state after **recover(TMNOFLAGS)** is called is unchanged.

Example 2: TMSTARTRSCAN Then TMNOFLAGS

When **recover(TMSTARTRSCAN)** is followed by **recover(TMNOFLAGS)**, the result is that all Xids are returned in **recover(TMSTARTRSCAN)** and no Xids are returned in subsequent **recover(TMNOFLAGS)** calls. The recovery scan state is **scanStarted**.

Example 3: TMSTARTRSCAN Already Called

When **recover(TMSTARTRSCAN)** has already been called—whether or not followed by **recover(TMNOFLAGS)**—and you then call **recover(TMENDRSCAN)**, the result is that all Xids are returned in **recover(TMSTARTRSCAN)** and no Xids are returned in the **recover(TMENDRSCAN)** call. The recovery scan state is **scanFinished**.

Example 4: Orphaned Branches

When any of the following have been called:

- **recover(TMSTARTRSCAN)** regardless of any previous **recover(...)** calls
- **recover(TMNOFLAGS)** after **recover(TMENDRSCAN)**
- **recover(TMENDRSCAN)** without calling **recover(TMSTARTRSCAN)** (This could throw **XAException**, but the implementation tolerates it.)

In all these cases, the result returned is the full set of prepared orphan branch's Xids.

You can set a flag to include both **TMSTARTRSCAN** and **TMENDRSCAN** (as allowed by X/Open XA spec) or a flag to include only **TMENDRSCAN**.

In either case, the recovery scan state is **scanFinished**.

Note: The SonicMQ administrative tools provide techniques for handling transactions that did not complete correctly by accessing the indoubt branches.

Distributed Transactions Models

Distributed transactions can allow:

- SonicMQ to participate in transactions controlled by an application server that involves other resources such as a database
- SonicMQ to participate in a distributed transaction
- One SonicMQ transaction be composed of PTP and Pub/Sub messages

SonicMQ Integrated with an Application Server

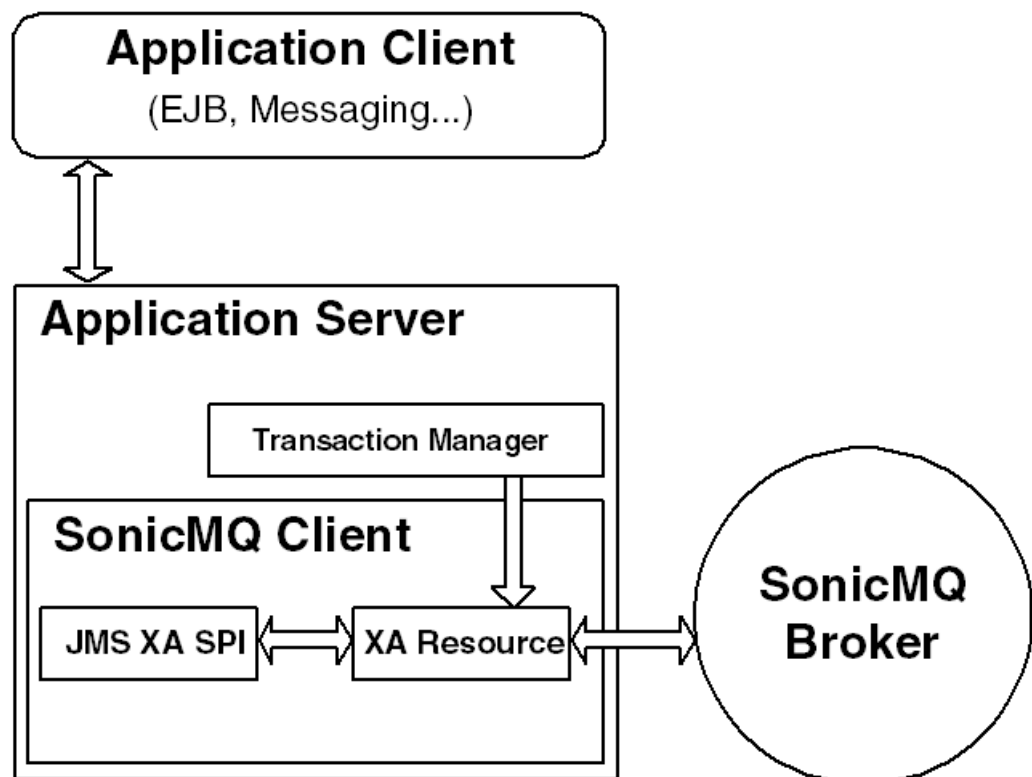
When a SonicMQ client is integrated with a JTA-capable application server, as shown in the following figure, the application server has a transaction manager. The application server uses JMS XA SPI, using the **XAConnectionFactory**, **XAConnection**, and **XASession** to get a reference to the **XAResource**, giving the **XAResource** to the Transaction Manager so that it begins, ends, prepares, and commits transaction work.

Inside SonicMQ, it is communicating with the SonicMQ server to let clients coordinate with the broker. This is transparent to the application server.

The Enterprise JavaBean (EJB) is not aware of the XA activity. The EJB only invokes **userTransaction.begin** and expects the application server to do the work.

The EJB might choose to use a container-managed transaction which means that everything is managed by the application server.

Figure 87: SonicMQ Integrated with an Application Server



Sample Code: Global Transaction When Integrated With Application Server

The following code snippet describes the significant work for performing a global transaction when SonicMQ is integrated with an application server. The sections show how to use a Message Driven Bean.

A Message Driven Bean (MDB) is in effect a message listener. The code appears as standard JMS methods that are intercepted by the application server. The transaction is managed by the application server which commits it when the **onMessage** terminates.

Using a Message Driven Bean

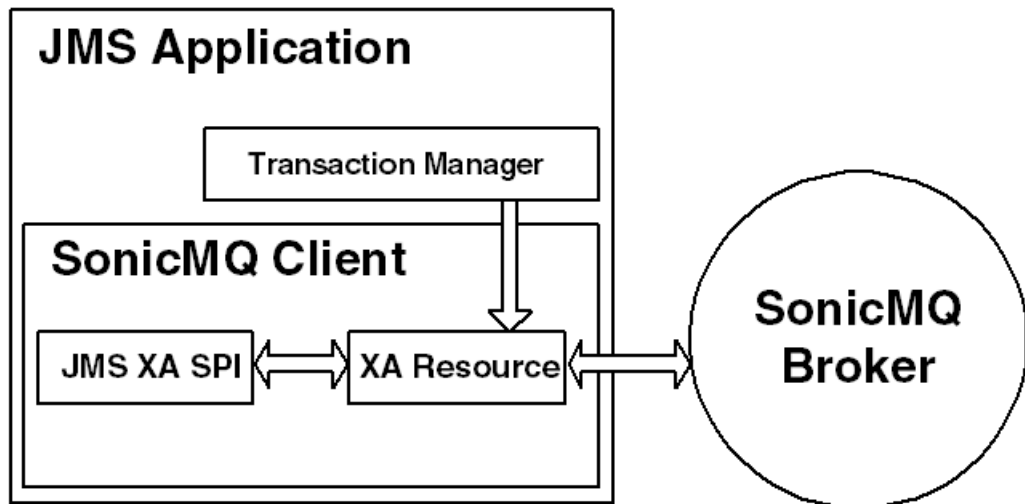
```
public void onMessage(Message message)
{ try
{
    InitialContext ctx = new InitialContext();
    ConnectionFactory cf
        =(ConnectionFactory)ctx.lookup("XAConnectionFactory");
    Connection c = cf.createConnection();
    Session s = c.createSession (false, Session.AUTO_ACKNOWLEDGE);
    Topic topic = s.createTopic("SampleT1");
    MessageProducer pub = s.createProducer( topic );
    TextMessage msgt = s.createTextMessage();
    pub.send(msgt);

    Queue queue = s.createQueue("SampleQ2");
    MessageProducer sender = s.createProducer( queue );
    TextMessage msgq = s.createTextMessage();
    sender.send(msgq);
    c.close();
    // ***** Oracle Connection *****
    DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/tebsource");
    java.sql.Connection dbConn = ds.getConnection();
    Statement s = dbConn.createStatement();
    s.executeUpdate ("insert into sonic_artist values('100','Bob Smith')");
    dbConn.close();
} catch(Exception e) {
    e.printStackTrace();
}
} //onMessage()
```

SonicMQ Directly Used with a Transaction Manager

When standing alone, a JMS application, as shown in the following figure, can use the JMS XA SPI to get an XA connection and an XA session that will get an XA resource reference for the transaction manager.

Figure 88: SonicMQ Directly Accessing a Transaction Manager



Sample Code: Global Transaction Using Transaction Manager

The following code describes the significant work for performing a global transaction using a transaction manager. After the application gets a reference to a Transaction Manager, the sections show how to:

- Produce messages in both domains under transactional control
- Complete the transaction across the session boundaries
- Shutdown the Object Request Broker (ORB)

Produce Messages in a Transaction

The TM is instructed to start a new transaction and associate it with the current thread:

```

...
try
{ tm.begin();
  Transaction txn = tm.getTransaction();

```

The transaction manager is instructed to provide the XAResources for the session:

```

XAConnectionFactory xacf =
    new progress.message.jclient.xa.XAConnectionFactory( url );
XAConnection xtc = xacf.createXAConnection( user, password );
Connection c = xtc;
XASession xas = xtc.createXASession();
XAResource xar = xas.getXAResource();

```

Then enlist the resource:

```

txn.enlistResource(xar);

```

Messages are produced in each session:

```
...
pub.send(msg1);
sender.send(msgA);
```

The resources are delisted:

```
...
txn.delistResource(xar, XAResource.TMSUCCESS);
} catch( Exception e )
{ tm.rollback();
}
```

Complete the Transaction

The transaction manager is instructed to start a two-phase commit:

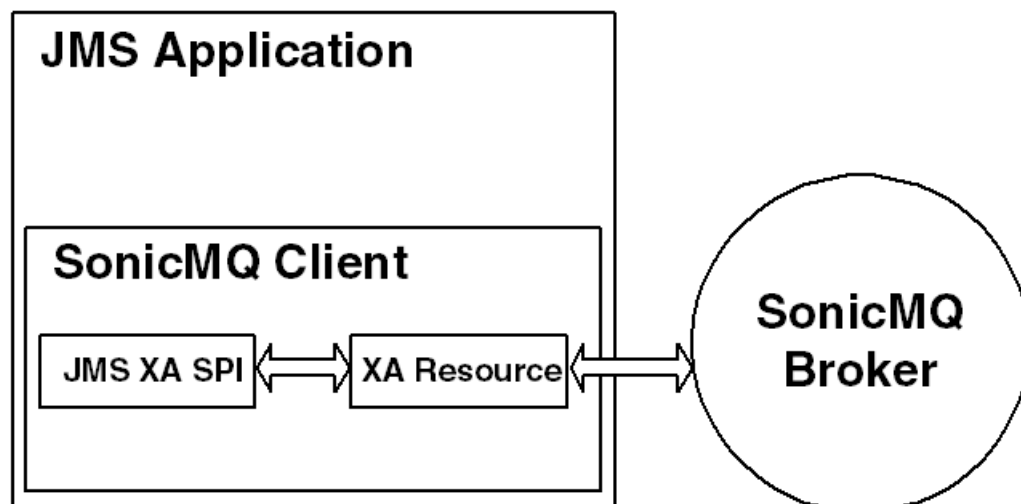
```
...
try
{ tm.commit();
} catch( Exception e )
{ e.printStackTrace();
}
```

At the application level, the transaction work is done. The transaction manager handles the preparation of the branches and the commit or rollback. If the prepares fail, the exception is thrown. If the commit succeeds, a return indicates successful commitment. If there are indoubt branches, the TM can work on the commit unless the TM or resource manager crashes. A TM recovery will resume the transaction work.

SonicMQ Performing DTP Without a Transaction Manager

Because the XA interface and the protocols are public, application developers can do all the work that is handled by the transaction manager directly in their code. The following figure shows the simple diagram where the application directly calls the JMS XA SPI and references the XA Resource. The developer will be responsible for the preparation and commit (or rollback) of the transaction.

Figure 89: SonicMQ Performing DTP Without a Transaction Manager



This constrains the application from the benefits of distributed transaction branches yet, for cases where two sessions, one PTP and one Pub/Sub, need to coordinate in a transaction, this model might be appropriate.

Sample Code: Global Transaction Without Transaction Manager

The following code describes the significant work for performing a global transaction without using a transaction manager. The sections show how to:

- Create the XAResource for a Point-to-point session
- Create the XAResource for a Publish and Subscribe session
- Produce messages in both sessions under transactional control
- Complete the transaction across session boundaries

Create XAResource for PTP

The **XAConnectionFactory** gets an **XAConnection**, which then gets a reference to a standard **Connection**. The **Connection** is the context for the JMS work:

```
...
XAConnectionFactory xaqc =
    new progress.message.jclient.xa.XAConnectionFactory ( url);

XAConnection xaqc = xaqc.createXAConnection(user, password);
Connection qc = xaqc;
```

An **XASession** is created, which then gets the **XAResource**. The methods of the **XAResource** are now accessible. A regular **Session** is created for the standard procedures of creating a queue and a sender in the context of the **XASession**:

```
XASession xaq = xaqc.createXASession();
XAResource xarq = xaq.getXAResource();
Session qs = xaq.getSession();
Queue q = qs.createQueue( "SampleQ1" );
MessageProducer sender = qs.createProducer( q );
```

Create XAResource for Pub/Sub

The Pub/Sub domain is set up in a similar way:

```
...
XAConnectionFactory xatcf =
    new progress.message.jclient.xa.XAConnectionFactory( url );
XAConnection xatc = xatcf.createXAConnection( user, password );

Connection tc = xatc;
XASession xats = xatc.createXASession();
XAResource xart = xats.getXAResource();
Session ts = xats.getSession();
Topic t = ts.createTopic( "SampleT1" );
MessageProducer pub = ts.createProducer( t );
...
```

Produce Messages

The transaction and its branches are distinguished in the sample code through the **xid**—an object that the application must create—whose first digit indicates the transaction identity and second digit represents the branch qualifier. The XAResource for both branches is started, messages are sent in both sessions, and then the XAResource calls the same identifiers to end the transaction branches:

```
...
try
{
    xarq.start(xidl1, XAResource.TMNOFLAGS);
    xart.start(xidl2, XAResource.TMNOFLAGS);
    ...
    pub.send(msg1);
}
```

```
sender.send(msgA);  
...  
xarq.end(xid11, XAResource.TMSUCCESS);  
xart.end(xid12, XAResource.TMSUCCESS);
```

If anything goes wrong, both branches roll back:

```
} catch( JMSEException e ) {  
  xarq.rollback(xid11);  
  xart.rollback(xid12);  
}
```

Complete the Transaction

To perform a two-phase commit, each branch is told to prepare. If they return successful, both branches are committed:

```
...  
try  
{ xarq.prepare(xid11);  
  xart.prepare(xid12);  
  xarq.commit(xid11, false);  
  xart.commit(xid12, false);
```

If either prepare failed, the transaction branches would roll back:

```
} catch( JMSEException e )  
{ xarq.rollback(xid11);  
  xart.rollback(xid12);  
}
```

If the first commit succeeded but the second failed, an in-doubt transaction would exist.

Running the Distributed Transaction Sample

The distributed transaction sample application demonstrates programmatic functions. A Producer console window establishes a distributed transaction, then publishes a message to a topic in one session and sends a message to a queue in another session. A Consumer console window participates in the distributed transaction with a topic subscriber and a queue receiver.

Note: The broker value defaults to **localhost:2506**. If your broker is at a remote location or on a different port, you must add the **-b broker:port** parameter. If security is enabled on the broker you must use different, established user names and include the **-p password** parameter with each user's password.

Be sure the SonicMQ container is running before executing any of the SonicMQ samples. See [Starting the SonicMQ Container and Management Console](#) for instructions about starting SonicMQ. For more detailed information on working with the Sonic Management Console, see the *Aurea SonicMQ Configuration and Management Guide*

Creating a distributed transaction

To create a distributed transaction:

1. Open a console window at the directory:
SonicMQ-install-dir\samples\DistributedTransaction\XASample

This will be your Producer window.

2. Enter the following command in the Producer window: `..\..\SonicMQ XASample -u Producer -qs SampleQ1 -tp SampleT1`

The sample starts and displays the command options when only producers—**qs** is the queue sender and **tp** is the topic publisher—are specified.

3. Type `START` and press **Enter**.

You are notified: **XA start xid1...**

4. Type text such as `One` and press **Enter**.
5. Type more text such as `Another` and press **Enter**.
6. Type `END` and press **Enter**.

You are notified: **XA end xid1...**

The messages are demarcated as global transaction **xid1**. A global transaction can now be started.

Creating receivers for a distributed transaction

To create receivers for a distributed transaction:

1. Open a console window at the directory:
`SonicMQ-install-dir\samples\DistributedTransaction\XASample`

This will be your Consumer window.

2. Type: `..\..\SonicMQ XASample -u Consumer -qr SampleQ1 -ts SampleT1`

The sample starts and displays the command options when only consumers—**qr** is the queue receiver and **ts** is the topic durable subscriber—are specified.

3. Type `STARTR` and press **Enter**.

You are notified: **xidR1 start ...**

4. Type `FROMQ` and press **Enter**.

Nothing happens. The messages that comprise the global send transaction have been demarcated but are not yet committed, and are unavailable to receivers.

Committing or rolling back the distributed transaction

To commit or rollback the distributed transaction:

In the Producer window, you can perform any of the following three actions to complete the transaction:

- Type `1PC` and press **Enter**.

You are notified: **XA commit xid1 DONE**

- Type `PREP` and press **Enter**.

You are notified: **XA prepare xid1 done!**

Then type `2PC` and press **Enter**.

You are notified: **XA commit xid1 DONE**

- If you want the transaction to terminate the transaction without actually sending any of its messages:

Type `ROLLBACK` and press **Enter**.

You are notified: **XA rollback xid1 DONE**

When you have committed a transaction, you can go to the Consumer window consume the messages.

Consuming messages from a distributed transaction

To consume messages from a distributed transaction:

1. In the Consumer window, type `FROMQ` and press **Enter**.

You receive the first message from the queue: **<Producer send> One.**

2. Type `FROMT` and press **Enter**.

You receive the first message from the topic: `DurableSubscriber: <Producer pub> One.`

3. Type `FROMQ` and press **Enter**.

You receive the second message from the queue: `<Producer send> Another.`

4. Type `FROMT` and press **Enter**.

You receive the second message from the topic: `DurableSubscriber: <Producer pub> Another.`

5. Type `ENDR` and press **Enter**.

You are notified: **xidR1 END**. This marks the end of the consumer's receiving of messages.

6. You can now close the Producer and Console Windows, and stop the SonicMQ container.

This sample application demonstrated a distributed transaction between a Producer and a Consumer. The Producer established the distributed transaction and then performed either a one- or two-phase commit to send the messages to a queue in one session, and to publish the messages to a topic in another session. The Consumer participated in the distributed transaction with both a queue receive and a topic subscriber. The Consumer was able to receive the message sent to the queue and the message sent to the topic.

Using the Sonic JNDI SPI

This appendix describes programming using the Sonic service provider implementation (SPI) for the Java Naming and Directory Interface (JNDI).

Important: Permission Denied Issues for Older Clients — If you are using JNDI SPI clients and your domain enforces management permissions (a feature introduced in V7.5), the JNDI SPI clients should be upgraded to at least V7.5 to avoid the potential of spurious **ConfigurePermissionsDenied** exceptions which could deny JNDI access.

For details, see the following topics:

- [Overview of the JNDI SPI](#)
- [Sonic JNDI SPI Samples](#)

Overview of the JNDI SPI

The Sonic JNDI SPI uses the Directory Service as the underlying store.

The Sonic SPI implements the **javax.naming.Context** interface. Management and client applications can use this interface to lookup and store JNDI compatible objects (serializable or referenceable), including SonicMQ JMS Administered object implementations for JMS connection factories and destinations.

The JNDI SPI can support two roles:

- An administrator who has read/write permission (the ability to create contexts and store objects)
- A read-only role that is limited to lookup and listing abilities

See the “Security” section in the chapter “JMS Administered Objects Tool” in the *Aurea SonicMQ Configuration and Management Guide* for information about the requirements and setup for these roles.

JNDI defines the way an initial context is obtained; obtaining a Sonic context follows these same techniques. Code Sample 27 provides a simple demonstration of JNDI programming with the Sonic SPI. The sample shows:

- Creating a JNDI environment (hash table) with Sonic SPI specific values and additional properties
- Obtaining an initial context
- Using the context to perform a lookup of a JMS Queue destination object

Programming with the Sonic JNDI SPI

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sonicsw.jndi.mfcontext.MFContextFactory");
env.put(Context.PROVIDER_URL, "tcp://localhost:2506");
env.put("com.sonicsw.jndi.mfcontext.domain", "Domain1");
env.put("com.sonicsw.jndi.mfcontext.idleTimeout", "60000");
env.put(Context.SECURITY_PRINCIPAL, "Administrator");
env.put(Context.SECURITY_CREDENTIALS, "Administrator");
Context ctx = new InitialContext(env);
..
Queue queue = (Queue)ctx.lookup("queues/Q1");
..
```

Note: The JNDI SPI and the environment properties do not have accompanying JavaDoc.

In certain applications and application servers, it is not possible to program the custom environment properties to configure the JNDI InitialContext. For these circumstances, Sonic supports parameter extensions to the connection URL in the following syntax:

```
protocol://host:port?domain=name&idleTimeout=int \
&requestTimeout=int&connectTimeout=int&node=name
```

The one or more parameter-value pairs can be in any order. For example:

```
tcp://localhost:2506?domain=Progress
tcp://localhost:2506?domain=Progress&idleTimeout=60000
tcp://localhost:2506?domain=Progress&connectTimeout=20000
tcp://localhost:2506?idleTimeout=60000&domain=Progress&connectTimeout=20000
```

Using Sonic-specific properties as parameters of the **PROVIDER_URL** would be as shown:

Using Sonic JNDI SPI with URL Parameters

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sonicsw.jndi.mfcontext.MFContextFactory");
env.put(Context.PROVIDER_URL,
        "tcp://localhost:2506?domain=Progress&idleTimeout=60000");
env.put(Context.SECURITY_PRINCIPAL, "Administrator");
env.put(Context.SECURITY_CREDENTIALS, "Administrator");
Context ctx = new InitialContext(env);
..
Queue queue = (Queue)ctx.lookup("queues/Q1");
..
```

Table 44 lists describes the values that should be assigned to the standard JNDI environment when using the Sonic SPI.

Table 44: Standard Environment Properties for the Sonic JNDI SPI (Continued)

Property (by constant name)	Description
javax.naming.Context. INITIAL_CONTEXT_FACTORY	The fully qualified class name of the Sonic SPI context factory that will be used to create the initial context ("com.sonicsw.jndi.mfcontext.MFContextFactory")
javax.naming.Context. PROVIDER_URL	The SonicMQ compliant URL string that describes how to connect to the Domain Manager broker(s) (default "tcp://localhost:2506")
javax.naming.Context. SECURITY_PRINCIPAL	The SonicMQ principal (username) that has permission to connect to the Domain Manager broker(s) and publish/subscribe to the messaging subjects used for JNDI communications with the Directory Service (see the "Security" section in the chapter "JMS Administered Objects Tool" in the <i>Aurea SonicMQ Configuration and Management Guide</i>). This property must be specified only if security is enabled on the management node.
javax.naming.Context. SECURITY_CREDENTIALS	The password associated with the SECURITY_PRINCIPAL.

Table 45 lists custom environment properties you can use with the Sonic JNDI SPI:

Table 45: Custom Environment Properties and URL Parameters for Sonic JNDI SPI (Continued)

Property (PROVIDER_URL parameter in bold)	Description
com.sonicsw.jndi.mfcontext.domain	The Sonic domain name to access (default "Domain1")
com.sonicsw.jndi.mfcontext.idleTimeout	A connection idle timeout in milliseconds (default value is "300000"— five minutes; the minimum value is "60000"— one minute). The connection to the Directory Service will be dropped if no activity takes place on the connection within the given timeout. If the connection has been dropped, it will be transparently reestablished on subsequent activity (you may notice a small delay).
com.sonicsw.jndi.mfcontext.requestTimeout	A request timeout in milliseconds (default "30000"). If a JNDI request—for example, bind, lookup, and so on—takes longer than the request timeout the request will fail. The requesting application will be thrown a javax.naming.NamingException with the root cause indicating a timeout condition. Depending on fault tolerance settings, the overall time before request failure may be some factor greater than the value of this property (as the request timeout will be applied to each fault tolerant option).

Property (PROVIDER_URL parameter in bold)	Description
com.sonicsw.jndi.mfcontext.connectTimeout	Elapsed time in milliseconds (default "10000"). The connect timeout is used to control the time to make an initial connection to the Domain Manager broker(s) and allowed time to reconnect in the event of a transient connection failure.
com.sonicsw.jndi.mfcontext.node	A Dynamic Routing Architecture (DRA) node through which JNDI communications with the Directory Service will be delegated (default none). Client applications may not be provided direct access to the Domain Manager broker(s), rather such communications will be delegated via other brokers using DRA.

The Sonic JNDI SPI sample applications provide examples of Java and JavaScript applications that use the SPI. The following sections discuss these sample applications.

Sonic JNDI SPI Samples

SonicMQ provides sample applications to help you get started with the JNDI API. These samples are included in your SonicMQ installation, and located in the management runtime samples directory at: *MQ_install_root\samples\Management\jndiAPI*

The following subdirectories provide Java and JavaScript samples:

- **java** — Java examples showing how to organize and store JMS administered objects in the Directory Service-based JNDI store provided with this SonicMQ installation
- **js** — javascript examples showing how to organize and store JMS administered objects in the Directory Service-based JNDI store provided with this SonicMQ installation

These directories contain scripts that run the samples or configure the environment in which to run the samples.

All samples directories have a **readme.txt** file that describes the contents more completely. Where appropriate, the **readme.txt** file contains instructions for running more complex samples.

Java JNDI SPI Sample

The Java-based Command Line Interface (CLI) sample is located in the directory *MQ_install_root\samples\Management\jndiAPI\java*. This sample demonstrates using the Sonic JNDI SPI. The application provides a simple command line interface that can be easily extended to add new, more specific commands for use in your applications.

This sample shows:

- Creating the initial context (and the environment setting required)
- Various typical calls on the context in order to manage Sonic JMS Administered objects

For more information about the features of this sample, see the comments in the file **JndiCLI.java**.

Important: To execute the CLI sample you must already have installed the SonicMQ container.

Running the CLI sample

To run the CLI sample:

1. Start the default broker for the container.
2. Open a command line console window, change directory to the following directory:
`MQ_install_root\samples\Management\jndiAPI\java`
3. Enter the appropriate command for your platform and administrative user:
 - For Windows, enter: `..\..\..\Mgmt JndiCLI Domain1 tcp://localhost:2506 Admin_user Admin_pwd`
 - For UNIX or Linux: `sh ../../../../Mgmt.sh JndiCLI Domain1 tcp://localhost:2506 Admin_user Admin_pwd`

Note: If you add a property on the command line and also as a parameter to the URL, the URL parameter will prevail. When using multiple parameters, some shells require that the complete URL statement be placed within quotes. For example: `>..\..\..\Mgmt JndiCLI Domain1 "tcp://localhost:2506?requestTimeout=60000&idleTimeout=100000" Administrator Administrator`

4. In the CLI shell you can get command help by typing a command followed by a question mark (?) character. For example: **> create context ?**

You can navigate around subcontext trees by moving a single subcontext at a time using the command **change context** (or **cc**).

The following CLI commands are supported by this sample application:

```
> {change context|cc} {/|..|<subcontext name>}
> {make context|mc} <subcontext name>
> {remove context|rc} <subcontext name>
> list
> lookup <name>
> bind tcf <name> <attribute=value>[,<attribute=value>...]
> bind qcf <name> <attribute=value>[,<attribute=value>...]
> bind topic <name> <topic name>
> bind queue <name> <queue name>
> unbind <name>
> exit
```

JavaScript JNDI API Samples

SonicMQ provides JavaScript JNDI API samples in the directory `MQ_install_root\samples\Management\jndiAPI\js`. This directory includes the following sample scripts:

- **List.js** — Demonstrates how to list the bindings in the root context of SonicMQ's JNDI SPI
- **CreateContext.js** — Demonstrates how to create a subcontext in the root context of SonicMQ's JNDI SPI
- **DestroyContext.js** — Demonstrates how to destroy a subcontext in the root context of SonicMQ's JNDI SPI
- **BindTopic.js** — Demonstrates how to bind a JMS Administered object (a **Topic**) in the root context of SonicMQ's JNDI SPI
- **UnbindTopic.js** — Demonstrates how to unbind a JMS Administered object (a **Topic**) in the root context of SonicMQ's JNDI SPI
- **Common.js** — Contains utility functions used by the samples in this directory

Each script assumes the installation default values were selected for domain name, container name, broker port, etc. If you made non-default selections during installation, you must modify the scripts appropriately.

Important: To execute the **JavaScript** samples you must already have installed the SonicMQ container.

Running the JavaScript proxy samples

To run the JavaScript proxy samples:

1. Start the default broker for the container.
2. Open a command line console window, change directory to the following directory:
`MQ_install_root\samples\Management\jndiAPI\js`
3. Enter the appropriate command for your platform:
 - For Windows, enter: `..\..\..\jsRun <scriptfile>`
 - For UNIX or Linux, enter: `sh ../../../../jsRun.sh <scriptfile>`

Note: Equivalent Java-based samples can be found in the directory:
`MQ_install_root\samples\Management\jndiAPI\java`

Using Client Tracing Logs

This appendix describes how SonicMQ Java clients can generate a log of key JMS API calls through aspect-oriented tracing of JMS API trace points. The functionality of aspect tracings are included in the SonicMQ development and client environments, and are easily enabled in the SonicMQ sample applications.

For details, see the following topics:

- [Overview of SonicMQ JMS API Tracing](#)
- [Enabling JMS Tracing](#)
- [Trace Levels](#)
- [Exploring Tracing in the SonicMQ Sample Applications](#)

Overview of SonicMQ JMS API Tracing

Aspects are useful during development of Java applications to facilitate debugging, testing and performance tuning of applications.

Traditionally you would code trace points in an application to expose a problem, and then remove those trace points when the application is ready for distribution. With AspectJ it is easy to both preserve the work of designing a good set of JMS trace points in a Java application, and disable the tracing when it is not being used. The tracing in SonicMQ is an aspect specifically for JMS tracing mode.

Java developers can use tracing to track an application's JMS usage patterns, and thereby detect issue such as leaking JMS artifacts, or not closing transactions. This section will describe the scope of tracings, the settings available, and how to implement tracing in the sample applications and the Sonic Diagnostics Framework.

Tracing has no overhead when it is not in use. Load-time weaving—adding the tracing code when the class is being loaded by the class loader—means that the aspect was not present when the application was compiled. So when load-time weaving is not configured, the application performs as if tracing code did not exist.

Even if you do not typically perform tracing functions, Aurea Sonic technical support might request that you run tracings to expose reported application problems.

See the Eclipse AspectJ web site, <http://www.eclipse.org/aspectj>, for more information.

Enabling JMS Tracing

The functionality of SonicMQ tracing is contained in three libraries distributed in the `<sonic_install_dir>/MQ10.0/lib` directory:

- `aspectjrt.jar`
- `aspectjweaver.jar`
- `sonic_tracing.jar`

When you put **sonic_tracing.jar** on an application's **CLASSPATH**, the classloader loads all three of these libraries.

The tracing aspect using AspectJ is enabled by configuring load-time weaving. Specify this in the following Java option: `-javaagent:%SONICMQ_LIB%\aspectjweaver.jar`

Trace Levels

There are four trace levels available:

Trace level	Tracing functions
0	<ul style="list-style-type: none"> None.
32	<ul style="list-style-type: none"> Exception tracing.
64	<ul style="list-style-type: none"> Exception tracing. Entry tracing.
128	<ul style="list-style-type: none"> Exception tracing. Entry tracing. Instance, argument and exit tracing

Setting the Trace Level in Applications

After you have enabled JMS tracing, set the trace level for an application with the Java system property **SonicMQ.DEBUG_NAME**. For example:

```
-DSonicMQ.DEBUG_NAME=tracing.jms:32
```

Exploring Tracing in the SonicMQ Sample Applications

SonicMQ's sample applications are located in the `sonic_install_dir/MQ10.0/samples` directory. The generic application startup scripts **SonicMQ.bat** (Windows®) and **SonicMQ.sh** (UNIX®/Linux®) reference the installation's environment settings and set up the command line to handle the application name and its parameters. Those files now include the required settings to implement tracing but the setting is commented out by default.

The tail of the file—shown here for Windows®—highlights the commented line you need to change:

```
...
rem
rem Uncomment the following set command to enable SonicMQ JMS API entry tracing.

rem set SONICMQ_SAMPLE_TRACE_LEVEL=64
if "%SONICMQ_SAMPLE_TRACE_LEVEL%"==" " (
    "%SONICMQ_JRE%"
    -cp ".;%SONICMQ_CLASSPATH%;%SOAP_CLASSPATH%"
    %SONICMQ_SSL_SAMPLES_CLIENT% %*
) else (
    "%SONICMQ_JRE%"
    -javaagent:"%WEAVER_JAR%"
    -cp ".;%TRACING_CLASSPATH%;%SONICMQ_CLASSPATH%;%SOAP_CLASSPATH%"
    %SONICMQ_SSL_SAMPLES_CLIENT%
    -DSonicMQ.DEBUG_NAME=tracing.jms:%SONICMQ_SAMPLE_TRACE_LEVEL% %*
```

Remove the comment mark (**rem**) from the line and save the file to enable all samples that use this script to implement and display tracings.

Using Tracing in the Sample Application Chat

The following code sections show how tracings display in a simple Hello world. use of the sample *sonic_install_dir*/MQ10.0/samples/TopicPubSub/Chat.

No Tracing (and Tracing Level 0)

Running the sample application without tracing, the output looks like this:

```
C:\Program Files\Aurea\Sonic\MQ10.0\samples\TopicPubSub\Chat>..\..\SonicMQ Chat
-u SALES
Enter text messages to clients that subscribe to the.jms.samples.chat topic.
Press Enter to publish each message.
Hello world.
SALES: Hello world.
Ctrl+C
```

This output would be the same when tracing is enabled at level 0. As this example threw no exceptions, the tracing level of 32 would generate similar output.

Exception Tracing

Setting tracing to level 32 displays exceptions as in this case of specifying a non-existent broker port:

```
C:\Program Files\Aurea\Sonic\MQ10.0\samples\TopicPubSub\Chat>
..\..\SonicMQ Chat -u SALES -b localhost:25006

[timestamp] exiting Connection
javax.jms.ConnectionFactory.createConnection(String, String)
exception: javax.jms.JMSEException: java.net.ConnectException: Connection refused:
  connect: localhost:25006
at progress.message.jimpl.JMSEExceptionUtil.createJMSEException
(JMSEExceptionUtil.java:42)
at progress.message.jimpl.Connection.<init>(Connection.java:954)
at progress.message.jclient.ConnectionFactory.createConnection
(ConnectionFactory.java:2140)
at Chat.chatter(Chat.java:53)
at Chat.main(Chat.java:225)
Caused by: [104] progress.message.client.ENetworkFailure:
java.net.ConnectException: Connection refused: connect: localhost:25006
at progress.message.zclient.Connection.
connectWithRecoveryOpt(Connection.java:1096)
... 12 more
```

Entry Tracing

The following shows the display when the **Chat** sample is set to trace level **64** and throws no exceptions:

```
C:\Program Files\Aurea\Sonic\MQ10.0\samples\TopicPubSub\Chat>
..\..\SonicMQ Chat -u SALES
[timestamp] entering Connection
javax.jms.ConnectionFactory.createConnection(String, String)
[timestamp] entering Session javax.jms.Connection.createSession(boolean, int)
[timestamp] entering Session javax.jms.Connection.createSession(boolean, int)
[timestamp] entering Topic javax.jms.Session.createTopic(String)
[timestamp] entering MessageConsumer javax.jms.Session.createConsumer(Destination)

[timestamp] entering void
javax.jms.MessageConsumer.setMessageListener(MessageListener)
[timestamp] entering MessageProducer javax.jms.Session.createProducer(Destination)

[timestamp] entering void javax.jms.Connection.start()
```

```

Enter text messages to clients that subscribe to the jms.samples.chat topic.
Press Enter to publish each message.
Hello world.
[timestamp] entering TextMessage javax.jms.Session.createTextMessage()
[timestamp] entering void javax.jms.TextMessage.setText(String)
[timestamp] entering void javax.jms.MessageProducer.send(Message)
[timestamp] entering void Chat.onMessage(Message)
[timestamp] entering String javax.jms.TextMessage.getText()
SALES: Hello world.
Ctrl+C
[timestamp] entering void javax.jms.Connection.close()

```

Instance, Argument and Exit Tracing

The following shows the display when the **Chat** sample is set to trace level **128** and throws no exceptions:

```

C:\Program Files\Aurea\Sonic\MQ10.0\samples\TopicPubSub\Chat>
..\..\SonicMQ Chat -u SALES
[timestamp] exiting progress.message.jclient.ConnectionFactory(String)
return: progress.message.jclient.ConnectionFactory@185572a
[timestamp] entering Connection
javax.jms.ConnectionFactory.createConnection(String, String)
instance: progress.message.jclient.ConnectionFactory@185572a
arguments: SALES, password
[timestamp] exiting Connection
javax.jms.ConnectionFactory.createConnection(String, String)
return: progress.message.jimpl.Connection@158291
[timestamp] entering Session javax.jms.Connection.createSession(boolean, int)
instance: progress.message.jimpl.Connection@158291
arguments: false, 1
[timestamp] exiting Session javax.jms.Connection.createSession(boolean, int)
return: jimpl.Session
[timestamp] entering Session javax.jms.Connection.createSession(boolean, int)
instance: progress.message.jimpl.Connection@158291
arguments: false, 1
[timestamp] exiting Session javax.jms.Connection.createSession(boolean, int)
return: jimpl.Session
[timestamp] entering Topic javax.jms.Session.createTopic(String)
instance: jimpl.Session
arguments: jms.samples.chat
[timestamp] exiting Topic javax.jms.Session.createTopic(String)
return: jms.samples.chat
[timestamp] entering MessageConsumer javax.jms.Session.createConsumer(Destination)
instance: jimpl.Session
arguments: jms.samples.chat
[timestamp] exiting MessageConsumer javax.jms.Session.createConsumer(Destination)
return: progress.message.jimpl.TopicSubscriber@f3552f
[timestamp] entering void
javax.jms.MessageConsumer.setMessageListener(MessageListener)
instance: progress.message.jimpl.TopicSubscriber@f3552f
arguments: Chat@f20434
[timestamp] exiting void
javax.jms.MessageConsumer.setMessageListener(MessageListener)
[timestamp] entering MessageProducer javax.jms.Session.createProducer(Destination)
instance: jimpl.Session
arguments: jms.samples.chat
[timestamp] exiting MessageProducer javax.jms.Session.createProducer(Destination)
return: progress.message.jimpl.TopicPublisher@1352367
[timestamp] entering void javax.jms.Connection.start()
instance: progress.message.jimpl.Connection@158291
[timestamp] exiting void javax.jms.Connection.start()
Enter text messages to clients that subscribe to the jms.samples.chat topic.
Press Enter to publish each message.
Hello world.

```

```
[timestamp] entering TextMessage javax.jms.Session.createTextMessage()
    instance:  jimpl.Session
[timestamp] exiting TextMessage javax.jms.Session.createTextMessage()
    return:    progress.message.jimpl.TextMessage@1cfd3b2
[timestamp] entering void javax.jms.TextMessage.setText(String)
    instance:  progress.message.jimpl.TextMessage@1cfd3b2
    arguments: SALES: Hello world.
[timestamp] exiting void javax.jms.TextMessage.setText(String)
[timestamp] entering void javax.jms.MessageProducer.send(Message)
    instance:  progress.message.jimpl.TopicPublisher@1352367
    arguments: progress.message.jimpl.TextMessage@1cfd3b2
[timestamp] exiting void javax.jms.MessageProducer.send(Message)
[timestamp] entering void Chat.onMessage(Message)
    instance:  Chat@f20434
    arguments: progress.message.jimpl.TextMessage@3727c5
[timestamp] entering String javax.jms.TextMessage.getText()
    instance:  progress.message.jimpl.TextMessage@3727c5
[timestamp] exiting String javax.jms.TextMessage.getText()
    return:    SALES: Hello world.

SALES: Hello world.
[timestamp] exiting void Chat.onMessage(Message)
Ctrl+C
[timestamp] entering void javax.jms.Connection.close()
    instance:  progress.message.jimpl.Connection@158291
```

Index

A

- access control lists
 - 28, 30, 176, 284
 - applied to multitopics 284
 - propagation of changed permissions 176
- ACID properties of a transaction 356
- acknowledgement mode
 - 167–169, 171, 250
 - AUTO_ACKNOWLEDGE 169
 - CLIENT_ACKNOWLEDGE 169
 - DUPS_OK_ACKNOWLEDGE 169
 - lazy 169
 - SINGLE_MESSAGE_ACKNOWLEDGE 169, 250
- active ping 131
- administered objects
 - 117, 124, 129, 177
 - ConnectionFactory 117
 - definition 124
 - destinations 177
 - readFile 129
- administrative notification 180
- ANSI C 32
- APIs
 - 54, 325, 371
 - JNDI 54, 371
 - SonicStream 325
- applet 32
- application
 - 186, 356
 - server 186, 356
- application/x-sonicmq-* 202
- asynchronous 223, 238
- authentication
 - 28, 30, 55, 111–112
 - consumer 30
 - in samples 55
 - producer 28
 - SSL
 - 111–112
 - client certificates 112
 - username and password 111
- authorization
 - 28, 30, 55
 - consumer 30
 - in samples 55
 - producer 28
- AUTO_ACKNOWLEDGE 169

B

- broker properties
 - 153
 - PREFERRED_ACTIVE 153
- broker storage
 - 157
 - NON_PERSISTENT_REPLICATED messages 157
- brokers
 - 34, 56–58, 132, 221, 346
 - failure 132
 - management
 - 221, 346
 - destination parameters 221
 - topic hierarchies 346
 - starting in UNIX 34, 57–58
 - starting in Windows 34, 56
- browsing queues
 - 70, 244
 - sample 70
- BytesMessage type 192

C

- C clients 32
- CAA-FF 153
- channel 308
- channel ID 309, 317
- characters
 - 52, 174, 217, 259, 261, 347, 349
 - reserved
 - 174, 217, 259, 261
 - in a Subscription name 261
 - in destination names 217
 - in topic names 174, 259
 - restricted 52
 - template 347, 349
- Chat sample application 59, 100
- chunks 328
- clearProperties 212
- client persistence
 - 88, 133, 149
 - fault-tolerant connections 149
 - under flow control 133
- client reconnect timeout 143
- CLIENT_ACKNOWLEDGE 169
- ClientPlus
 - 68, 88, 133, 307
 - client persistence 133
 - large message support 307

- clients
 - [119, 167](#)
 - identifier [119](#)
 - session [167](#)
 - clusters
 - [23, 274, 277](#)
 - behavior of shared subscriptions [274](#)
 - global subscriptions [277](#)
 - commit
 - [75, 171, 357](#)
 - in a global transaction [357](#)
 - communication resource manager [356](#)
 - compiling samples [58, 100](#)
 - ConnectionFactorys
 - [117, 124](#)
 - administered objects [124](#)
 - definition [117](#)
 - connections
 - [26, 79, 108, 119, 140, 163–164, 187](#)
 - consumer [187](#)
 - definition [26](#)
 - fault-tolerant [140](#)
 - identifier [119](#)
 - multiple [164](#)
 - retry when broken [79](#)
 - starting, stopping, closing [163](#)
 - consumers
 - [187](#)
 - connection [187](#)
 - content ID [202](#)
 - content type [201–202](#)
 - Continuous Availability
 - [140](#)
 - client connection [140](#)
 - CorrelationID
 - [91, 206, 219](#)
 - sample [91](#)
 - count, maximum delivery attempts [122–123, 170](#)
 - count, prefetch [243](#)
 - createBrowser [244](#)
 - createDurableSubscriber [261](#)
 - createMessage [193, 260](#)
 - createQueue [174](#)
 - createQueueConnection [130](#)
 - createStream [329](#)
 - createSubscriber [260](#)
 - createTopic [174](#)
- ## D
- DataHandler [197, 201](#)
 - dead message [289](#)
 - dead message queue
 - [31, 79, 84, 198, 203, 222, 246, 287, 289–291, 293, 295–296](#)
 - default properties [291](#)
 - dead message queue (*continued*)
 - enabling features [291](#)
 - full [295](#)
 - monitoring [291](#)
 - notification factor [291](#)
 - persistence [222](#)
 - programming [246](#)
 - QoS level [31](#)
 - reason code [84, 203](#)
 - sample [79](#)
 - specifying a preferred destination [296](#)
 - system [290](#)
 - using MultipartMessage [203](#)
 - wrapping problem messages [198](#)
 - DeadMessages sample application [79](#)
 - delivery mode
 - [153, 205–206, 220, 222, 260, 289](#)
 - default value [205](#)
 - message header field [206](#)
 - NON_PERSISTENT [289](#)
 - NON_PERSISTENT_REPLICATED [153](#)
 - on the broker [222](#)
 - PERSISTENT [220, 222, 289](#)
 - producer parameter [260](#)
 - delivery mode (SonicStream API) [328](#)
 - destination factory
 - [281](#)
 - creating a multitopic [281](#)
 - destinations
 - [91, 124, 177, 230](#)
 - administered objects [124, 177](#)
 - temporary [91, 230](#)
 - Distributed Transaction sample application [368](#)
 - distributed transactions [172](#)
 - DMQ
 - [289](#)
 - See dead message queue [289](#)
 - Document Object Model [64, 103, 194–195](#)
 - DocumentBuilder [194](#)
 - dropped connection
 - [77, 132](#)
 - sample [77](#)
 - duplicate message detection [172](#)
 - DUPS_OK_ACKNOWLEDGE [169](#)
 - durable subscriptions
 - [46, 86, 222, 261, 284](#)
 - creating [46](#)
 - definition [261](#)
 - handling on the broker [222](#)
 - multitopic [284](#)
 - sample [86](#)
 - unsubscribing [261](#)
 - DurableChat sample application [86](#)
 - dynamic routing architecture
 - [79, 302](#)
 - undelivered reason codes [302](#)

dynamic routing architecture (*continued*)
 with a dead message queue [79](#)

E

encryption
 [28](#), [208](#), [220](#)
 per message [208](#), [220](#)
 enumeration for queue browsing [244](#)
 events
 [180](#), [246](#)
 flow control [180](#)
 notify undelivered [246](#)
 expiration
 [30](#), [207](#), [220](#), [222](#), [248](#)
 QoS level [30](#)
 expired message [289–290](#)
 extended type [198](#), [208](#)

F

FastForward [153](#)
 fault tolerant client [140](#)
 fault-tolerant connections
 [140](#), [149](#), [156](#)
 client persistence [149](#)
 NON_PERSISTENT_REPLICATED delivery mode
 [156](#)
 file transfers [68](#)
 filters [224](#)
 flow control
 [133](#), [179–180](#), [182](#), [275](#), [285](#)
 disabling [182](#)
 events [180](#)
 multitopics [285](#)
 producers using client persistence [133](#)
 shared subscriptions [275](#)
 flow to disk [182](#)
 fragments [309](#)
 FT_REPLICATE_NON_PERSISTENT [153](#)

G

getPropertyNames [212](#)
 global subscription
 [277](#), [283](#)
 multitopic [283](#)
 shared [277](#)
 group ID [211](#)
 group identifier [95](#)
 guaranteeing delivery [291](#)

H

header message [307](#)

headers
 [204–205](#)
 default header field values [205](#)
 message [205](#)
 SOAP [204](#)
 hierarchical name spaces
 [261](#), [345](#)
 as message filters [261](#)
 HierarchicalChat sample application [98](#)
 host
 [59](#)
 remote [59](#)
 hostname [118](#)
 HTTP Direct
 [285](#)
 multitopic [285](#)
 HTTP tunneling [113](#)
 HTTPS [114](#)

I

identifier
 [119](#)
 client [119](#)
 connection [119](#)
 indoubt
 [303](#), [357](#)
 messages [303](#)
 transaction state [357](#)
 indoubt message [290](#)
 initial connect timeout [121](#), [143](#)
 instanceof operator [68](#)

J

J2EE [22](#), [188](#)
 Java
 [31–32](#), [356](#), [359](#)
 applet [32](#)
 client [31](#)
 Transaction API [359](#)
 Transaction Service [356](#)
 JAXP [194](#)
 JMS provider [24](#)
 JMS_SonicMQ message properties [292](#)
 JMS_SonicMQ_ExtendedType [198](#)
 JMSX properties [210](#)
 JMSXDeliveryCount [170](#), [210](#)
 JMSXUserID [210](#)
 JNDI
 [177](#), [218](#), [259](#)
 lookup of destinations [177](#)
 lookup of topics [218](#), [259](#)
 JNDI SPI [54](#), [371](#)
 JNDI SPI sample applications [374](#)

L

- large message support [68](#), [307](#)
- latency [217](#)
- lazy acknowledgement [169](#)
- listeners
 - [134](#), [224](#), [239](#), [317](#), [329](#)
 - channel [317](#)
 - message [239](#)
 - rejection [134](#)
 - SonicStream
 - [329](#)
 - exception [329](#)
 - notification [329](#)
- local store [133](#)
- LocalStore sample application (PTP) [90](#)
- LocalStore sample application (Pub/Sub) [90](#)
- log
 - [133](#), [307](#)
 - local store [133](#), [307](#)
- loop test [100](#)

M

- management APIs
 - [54](#), [371](#)
 - JNDI SPI [54](#), [371](#)
- MapMessage
 - [63](#), [102](#), [192](#)
 - enhancing the sample [102](#)
 - sample application [63](#), [102](#)
 - type [192](#)
- MaxDeliveryCount [303](#)
- message
 - [94–95](#), [103](#), [173](#), [192](#), [197](#), [208](#), [213](#), [217](#), [220](#), [222](#), [224](#), [237–238](#), [244](#), [258](#), [289–290](#), [292](#), [294–295](#), [302](#), [307](#), [309](#)
 - body
 - [103](#), [213](#)
 - setting and getting [213](#)
 - Text [103](#)
 - XML (DOM format) [103](#)
 - dead [289](#)
 - delivery
 - [238](#)
 - PTP [238](#)
 - expired [289–290](#)
 - file fragments [309](#)
 - groups
 - [95](#)
 - sample [95](#)
 - indoubt [290](#)
 - JMS_SonicMQ properties [292](#)
 - large [307](#)
 - NON_PERSISTENT [289](#)

- message (*continued*)
 - ordering
 - [217](#), [237](#), [258](#)
 - PTP [237](#)
 - Pub/Sub [258](#)
 - parts [197](#)
 - PERSISTENT [220](#), [222](#), [289](#)
 - properties [208](#)
 - reliability
 - [217](#), [238](#), [258](#)
 - PTP [238](#)
 - Pub/Sub [258](#)
 - selectors
 - [94](#), [224](#), [244](#)
 - on QueueBrowser [244](#)
 - on server for topics, option [224](#)
 - sample [94](#)
 - types [173](#), [192](#)
 - undeliverable [289](#)
 - undelivered
 - [290](#), [294–295](#), [302](#)
 - handling [294](#)
 - types [295](#)
 - unroutable [290](#)
- Message Driven Beans [188](#)
- message property
 - [208](#)
 - maximum length [208](#)
- message selectors
 - [225](#)
 - maximum length [225](#)
- Message type [192](#)
- MessageGroupTalk sample application [95](#)
- MessageID [206](#)
- MessageMonitor sample application [72](#)
- MessagePart [197](#), [202](#)
- MIME [201](#)
- monitoring interval [180](#)
- MultipartMessage type [192](#), [197](#)
- multitopic [280](#)
- MultiTopicChat sample application [61](#)

N

- name spaces
 - [195](#), [345](#)
 - NamespaceAware [195](#)
- network failure [132](#)
- noLocal [260–261](#)
- NON_PERSISTENT
 - [153](#), [289](#)
 - message [289](#)
 - upgrade to REPLICATED [153](#)
- NON_PERSISTENT_REPLICATED [153](#)
- notification factor [291](#)
- notification topic (SonicStream API) [327](#)

notify undelivered [208](#), [248](#)
 NoWait [223](#), [239](#)
 null
 [227](#), [348](#)
 in comparison tests [227](#)
 in topic naming [348](#)

O

object model [26](#)
 ObjectMessage type [192](#)
 one-to-many [24](#)
 one-to-one [24](#)

P

parts [197](#)
 peer-to-peer [68](#), [307](#)
 pending queue [211](#)
 persistence
 [29](#), [88](#), [133](#), [206](#), [222](#), [307](#)
 local [88](#), [133](#), [307](#)
 message delivery mode [206](#)
 on the broker [222](#)
 QoS options [29](#)
 PERSISTENT
 [220](#), [222](#), [289](#)
 message [220](#), [222](#), [289](#)
 ping interval [131](#)
 point-to-point [24](#)
 poison message scenario
 [170](#)
 handling by limiting redelivery [170](#)
 port [118](#)
 PREFERRED_ACTIVE broker property [153](#)
 prefetch
 [243](#)
 count [243](#)
 threshold [243](#)
 preserve undelivered [208](#)
 priority
 [30](#), [205](#), [207](#), [222](#), [260](#)
 default value [205](#)
 header field [207](#)
 on the broker [222](#)
 publish parameter [260](#)
 QoS level [30](#)
 producers [26](#), [218](#)
 properties
 [84](#)
 dead message [84](#)
 propertyExists [212](#)
 protocols [110](#), [118](#)
 proxy servers [114](#)
 publish
 [205](#), [260](#), [280](#)

publish (*continued*)
 multitopic [280](#)
 publish and subscribe [24](#)
 publishers [218](#), [259](#)

Q

QoP cache size [284](#)
 quality of protection
 [28](#), [284](#)
 applied to multitopics [284](#)
 quality of service
 [28](#), [77](#), [88](#)
 sample
 [77](#), [88](#)
 client persistence [88](#)
 durable subscription [77](#)
 persistent storage [77](#)
 reliable connection [77](#)
 queue
 [293](#)
 dead messages [293](#)
 QueueMonitor sample application [70](#)
 queues
 [40](#), [70](#), [236](#), [238](#), [244](#)
 browser [40](#), [244](#)
 browser sample [70](#)
 listener [238](#)
 set up [236](#)

R

readAheadWindow (SonicStream API) [331](#)
 readFile [129](#)
 reason codes [302](#)
 receivers
 [36](#), [40](#), [223](#), [238–239](#)
 message selectors [36](#), [40](#)
 multiple [238](#)
 reconnect
 [134](#)
 in client persistence [134](#)
 recoverable file channel [68](#), [307](#)
 recovery
 [133](#), [307](#), [318](#)
 client local stores [133](#), [307](#)
 file channel [318](#)
 redelivered [29](#), [207](#)
 redelivery
 [156](#)
 NON_PERSISTENT_REPLICATED [156](#)
 redelivery limit [122](#), [170](#)
 RejectionListener [134](#)
 rejections [135](#)
 releaseStream [329](#)
 ReliableTalk sample application [78](#), [89–90](#)

- remote host [59](#)
- remote publishing
 - [277](#), [283](#)
 - global subscriptions [277](#)
 - multitopic [283](#)
 - shared subscriptions [277](#)
- remote subscribing (global) [277](#)
- ReplyTo [206](#), [219](#)
- request and reply
 - [30–31](#), [230](#)
 - QoS level [30–31](#)
- Request and Reply sample application (PTP) [92](#)
- Request and Reply sample application (Pub/Sub) [92](#)
- requestor [92](#)
- resources
 - [356–357](#)
 - enlistment [357](#)
 - manager [356](#)
- restricted characters [52](#)
- retry count [309](#)
- retry interval [309](#)
- rollback
 - [75](#), [171](#)
 - definition [171](#)
- RoundTrip sample application [100–101](#)
- routing
 - [303](#)
 - problems causing non-delivery [303](#)
- routing node [23](#)
- routing statistics [211](#)
- RSA Security [110](#)

S

- samples
 - [59–61](#), [63–66](#), [70](#), [72](#), [74–75](#), [78–79](#), [86](#), [88–90](#), [92](#), [94–95](#), [98](#), [100–103](#), [333](#), [368](#), [374](#)
 - Chat (Pub/Sub)
 - [59](#), [100](#)
 - extended for common topics [100](#)
 - Distributed Transaction [368](#)
 - DurableChat (Pub/Sub)
 - [86](#), [100](#)
 - extended for common topics [100](#)
 - HierarchicalChat (Pub/Sub) [98](#)
 - JNDI SPI [374](#)
 - LocalStore (PTP) [88](#), [90](#)
 - LocalStore (Pub/Sub) [90](#)
 - MapMessage (PTP) [63](#)
 - MapMessages (PTP)
 - [102](#)
 - extended for other data types [102](#)
 - MessageGroupTalk(PTP)) [95](#)
 - MessageMonitor (Pub/Sub) [72](#)
 - MultiTopicChat (Pub/Sub) [61](#)
 - QueueMonitor (PTP) [70](#)
- samples (*continued*)
 - ReliableTalk (PTP) [78](#), [89–90](#)
 - Request and Reply (PTP) [92](#)
 - Request and Reply (Pub/Sub) [92](#)
 - RoundTrip (PTP)
 - [100–101](#)
 - extended for various behaviors [101](#)
 - SelectorChat (Pub/Sub) [94](#)
 - SelectorTalk (PTP) [94](#)
 - SonicStreams
 - [333](#)
 - StreamReceiver [333](#)
 - StreamSender [333](#)
 - Talk (PTP) [60](#)
 - Transacted Messages (PTP) [74](#)
 - Transacted Messages (Pub/Sub) [75](#)
 - using DeadMessages application (PTP) [79](#)
 - XA [368](#)
 - XMLChat (Pub/Sub) [64](#)
 - XMLDOMChat (Pub/Sub) [66](#)
 - XMLMessage (PTP) [65](#)
 - XMLMessage (Pub/Sub)
 - [103](#)
 - extended with additional data [103](#)
 - XMLSAXChat(PubSub) [66](#)
 - XMLSAXTalk (PTP) [65](#)
 - XMLTalk (PTP) [64](#)
- SAX [194](#), [196](#)
- SAX parser [194](#)
- SAX XML parser [64](#)
- scripts
 - [58](#)
 - batch files [58](#)
 - for compiling modified samples [58](#)
 - for running samples [58](#)
 - shell scripts [58](#)
- security
 - [55](#), [346](#)
 - in samples [55](#)
 - in topic name spaces [346](#)
- segment (SonicStream API) [328](#)
- selector string [94](#)
- SelectorChat sample application [94](#)
- selectors
 - [225](#)
 - maximum length [225](#)
- SelectorTalk sample application [94](#)
- send [205](#)
- serialized Java objects [129](#)
- server session pool [187](#)
- sessions
 - [26](#), [35](#), [43](#), [108](#), [167](#), [171](#), [173](#), [187](#)
 - client [167](#)
 - definition [26](#), [108](#), [167](#)
 - multiple [167](#)
 - objects [173](#)

- sessions (*continued*)
 - pool [187](#)
 - queue [35](#)
 - topic [43](#)
 - transacted [171](#)
 - shared subscriptions
 - [266](#), [275](#), [285](#)
 - flow control [275](#)
 - flow to disk [275](#)
 - multitopic [285](#)
 - SINGLE_MESSAGE_ACKNOWLEDGE [169](#), [250](#)
 - SOAP [67](#), [197](#), [204](#)
 - socket connect timeout [121](#), [143](#)
 - SonicStream API [325](#)
 - split delivery (multitopic) [282](#)
 - SQL [94](#)
 - SQL92 [224](#)
 - SSL
 - [110](#)
 - JSSE [110](#)
 - RSA [110](#)
 - status
 - [314](#)
 - channel [314](#)
 - store directory [134](#)
 - stream topic [326](#)
 - StreamMessage type [192](#)
 - subscribe
 - [280](#)
 - multitopic [280](#)
 - subscribers
 - [46](#), [260](#)
 - definition [260](#)
 - message selectors [46](#)
 - No Local Delivery [46](#)
 - subscriptions
 - [46](#), [266](#)
 - durable [46](#)
 - shared [266](#)
 - support, technical [20](#)
 - synchronous [223](#), [238](#)
 - syntax
 - [225](#), [346](#)
 - message selector string [225](#)
 - topic names [346](#)
 - system dead message queue [290](#)
- T**
- Talk sample application [60](#)
 - TCP [110](#)
 - TCP_RESET [132](#)
 - technical support [20](#)
 - template characters
 - [218](#), [259](#), [347](#), [349](#)
 - topics [218](#), [259](#)
 - temporary destination [91](#), [230](#)
 - TextMessage type [192](#)
 - threads [211](#)
 - threshold, prefetch [243](#)
 - time to live [295](#)
 - time-to-live
 - [86](#), [205](#), [208](#), [222](#), [260](#)
 - default value [205](#)
 - DurableChat sample [86](#)
 - message property [208](#)
 - on the broker [222](#)
 - publish parameter [260](#)
 - timeout
 - [121](#), [134](#), [143](#), [172](#), [223](#), [239](#), [309](#)
 - client reconnect [143](#)
 - in client persistence [134](#)
 - in transacted sessions [172](#)
 - initial connect [121](#), [143](#)
 - on a file channel sender [309](#)
 - on a receiver [223](#)
 - on receiveNoWait [223](#), [239](#)
 - on synchronous receive [239](#)
 - socket connect [121](#), [143](#)
 - timestamp
 - [206](#), [208](#)
 - undelivered [208](#)
 - TopicRequestor [231](#)
 - topics
 - [100](#), [218](#), [259](#), [345](#)
 - common in samples [100](#)
 - definition [259](#)
 - hierarchical name spaces [218](#), [259](#)
 - hierarchy [345](#)
 - transacted sessions
 - [167–168](#), [171](#)
 - definition [171](#)
 - session parameter [167–168](#)
 - TransactedChat sample application [75](#)
 - TransactedTalk sample application [74](#)
 - transactions
 - [157](#), [172](#), [356–357](#)
 - context [357](#)
 - distributed [172](#)
 - effect of NON_PERSISTENT_REPLICATED [157](#)
 - global [356](#)
 - local [356](#)
 - transaction manager [356](#)
 - TTL
 - [295](#)
 - See time to live [295](#)
 - two-phase commit [357](#)
 - type [207](#), [219](#)
- U**
- undeliverable message [289](#)

- undelivered
 - [31, 81, 208, 248, 302](#)
 - notify [31, 81, 208](#)
 - preserve [31, 81, 208](#)
 - reason codes [208, 248, 302](#)
 - timestamp [208](#)
- undelivered destination
 - [299](#)
 - message properties [299](#)
- undelivered message
 - [290, 294–295, 297](#)
 - handling [294](#)
 - override
 - [297](#)
 - destination name [297](#)
 - destination type [297](#)
 - types [295](#)
- undelivered message reason codes [302](#)
- UNDELIVERED_DELIVERY_LIMIT_EXCEEDED [170, 303](#)
- unfinished channel [318](#)
- unroutable message [290](#)
- unsubscribe [261](#)
- URL [118](#)
- username [119](#)
- UUID [172](#)

V

- valueOf method [213](#)

W

- wildcards [99](#)
- window size [309](#)

X

- X-HTTP-* properties [211](#)
- XA sample application [368](#)
- XAResource [356–357](#)
- XID [357](#)
- XML message
 - [64, 103, 178](#)
 - create method [178](#)
 - enhanced sample [103](#)
 - sample application [64](#)
- XML parser [64](#)
- XMLMessage sample application (PTP) [65](#)
- XMLMessage sample application (Pub/Sub) [103](#)
- XMLMessage type [192, 194](#)
- XMLSAXChat sample application [66](#)
- XMLSAXTalk sample application [65](#)
- XMLTalk sample application [64](#)