Progress Sonic™

SonicMQ® Product Family

Getting Started with Progress SonicMQ V7.5

PROGRESS SOFTWARE

Getting Started with Progress SonicMQ V7.5

Copyright © 2007 Sonic Software Corporation. All rights reserved. Sonic Software Corporation is a wholly-owned subsidiary of Progress Software Corporation.

The Sonic Software products referred to in this document are also copyrighted, and all rights are reserved by Sonic Software Corporation and/or its licensors, if any. This manual may not, in whole or in part, be copied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Sonic Software Corporation.

The information in this manual is subject to change without notice, and Sonic Software Corporation assumes no responsibility for any errors that may appear in this document. The references in this manual to specific platforms supported are subject to change.

Dynamic Routing Architecture, Sonic ESB, SonicMQ, Sonic Software (and design), Sonic Orchestration Server, and SonicSynergy are registered trademarks of Sonic Software Corporation in the U.S. and other countries. Connect Everything. Achieve Anything., Sonic SOA Suite, Sonic Business Integration Suite, Sonic Collaboration Server, Sonic Continuous Availability Architecture, Sonic Database Service, Sonic eXtensible Information Server, Sonic Workbench, and Sonic XML Server are trademarks of Sonic Software Corporation in the U.S. and other countries. Progress is a registered trademark of Progress Software Corporation in the U.S. and other countries. IBM is a registered trademark of IBM Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Any other trademarks or service marks contained herein are the property of their respective owners.

SonicMQ Product Family includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

SonicMQ Product Family includes code licensed from Mort Bay Consulting Pty. Ltd. The Jetty Package is Copyright © 1998 Mort Bay Consulting Pty. Ltd. (Australia) and others.

SonicMQ Product Family includes the JMX Technology from Sun Microsystems, Inc. Use and Distribution is subject to the Sun Community Source License available at http://sun.com/software/communitysource.

SonicMQ Product Family includes software developed by the ModelObjects Group (http://www.modelobjects.com). Copyright 2000-2001 ModelObjects Group. All rights reserved. The name "ModelObjects" must not be used to endorse or promote products derived from this

software without prior written permission. Products derived from this software may not be called "ModelObjects", nor may "ModelObjects" appear in their name, without prior written permission. For written permission, please contact djacobs@modelobjects.com.

SonicMQ Product Family includes files that are subject to the Netscape Public License Version 1.1 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/NPL/. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is Mozilla Communicator client code, released March 31, 1998. The Initial Developer of the Original Code is Netscape Communications Corporation. Portions created by Netscape are Copyright 1998-1999 Netscape Communications Corporation. All Rights Reserved.

SonicMQ Product Family includes a version of the Saxon XSLT and XQuery Processor from Saxonica Limited that has been modified by Progress Software Corporation. The contents of the Saxon source code and the modified source code file (Configuration.java) are subject to the Mozilla Public License Version 1.0 (the "License"); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ and a copy of the license (MPL-1.0.html) can also be found in the installation directory, in the Docs7.5/third_party_licenses folder, along with a copy of the modified code (Configuration.java); and a description of the modifications can be found in the Progress SonicMQ v7.5 README file. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is The SAXON XSLT and XQuery Processor from Saxonica Limited. The Initial Developer of the Original Code is Michael Kay http://www.saxonica.com/products.html). Portions created by Michael Kay are Copyright © 2001-2005. All rights reserved. Portions created by Progress Software Corporation are Copyright © 2007. All rights reserved.

SonicMQ Product Family includes software developed by Apache Software Foundation (http://www.apache.org/). Copyright © 1999-2000 The Apache Software Foundation. All rights reserved. The names "Ant," "Axis," "Xalan," "FOP," "The Jakarta Project," "Tomcat," "Xerces" and/or "Apache Software Foundation" must not be used to endorse or promote products derived from the Product without prior written permission. Any product derived from the Product may not be called "Apache", nor may "Apache" appear in their name, without prior written permission. For written permission, please contact apache@apache.org.

SonicMQ Product Family includes software Copyright © 1999 CERN - European Organization for Nuclear Research. Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. CERN makes no representations about the suitability of this software for any purpose. It is provided "as is" without expressed or implied warranty.
SonicMQ Product Family includes software developed by the University Corporation for Advanced Internet Development <http://www.ucaid.edu>Internet2 Project. Copyright © 2002 University Corporation for Advanced Internet Development, Inc. All rights reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used to endorse or promote products derived from this software and products derived from this software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for Advanced Internet Development, nor may OpenSAML appear in their name without prior written permission of the University Corporation for Advanced Internet Development. For written permission, please contact opensaml@opensaml.org.

April 2007

Contents

Preface 9
About This Guide 9
How to Use This Guide 9
Typographical Conventions 10
SonicMQ Documentation 11
Worldwide Technical Support 13

Chapter 1: Introducing SonicMQ 15
Introduction 15
What about RPC-Based Mechanisms? 17
What Problems Does Messaging Solve? 19
Disparate System Integration 19
Exchanging Information With Business Partners 19
Automating Common Business Functions 20
Challenges of Building Next Generation Applications 21
SonicMQ: The Software for Mission-Critical Messaging 23
Guaranteed Reliability 23
Very High Performance 24
Massive Scalability 24
High Availability 24
End-to-end Security 25
Management Efficiency 25

Chapter 2: SonicMQ Messaging . . . 27

Messaging Concepts . . . 27
- Messaging Models . . . 27
- Unified Messaging . . . 28
- Publish and Subscribe Messaging: Broadcast the Message . . . 34
- Point-to-point Messaging: Each Message Has One Consumer . . . 36
- SonicMQ Messaging Clients . . . 38

SonicMQ Features for Developing Client Applications . . . 39
- Features for Publishers and Subscribers . . . 39
- Flow Control . . . 40
- Option to Preserve Undelivered Messages . . . 40
- Dynamic Routing Architecture . . . 40
- Security . . . 41
- Duplicate Message Detection . . . 41
- Recoverable File Channels . . . 41
- Client Persistence . . . 42

Chapter 3: SonicMQ Sample Applications . . . 43

Getting Started With Selected Samples . . . 43
- Publish and Subscribe (TopicPubSub Folder) . . . 43
- Queue Point-to-point (QueuePTP Folder) . . . 44

Installing SonicMQ to Run the Sample Applications . . . 45
- After the Installation Is Complete . . . 46
- Starting the Broker . . . 47
- Preparing to Run the SonicMQ Samples . . . 47

Topic Publish and Subscribe Samples . . . 49
- Chat Application . . . 50
- Message Monitor . . . 52
- Durable Chat Application . . . 53
- Shared Subscriptions . . . 56
- Selector Chat Application . . . 57
- Hierarchical Chat Application . . . 58
- XML Messages . . . 59

Queue Point-to-point Samples . . . 62
- About Queues . . . 62
- Talk Application . . . 63
- Queue Monitor . . . 64
- Reliable Talk Application . . . 66
- Working with MultiPart Messages . . . 66

Transacted Talk Sessions 68
Request and Reply 69
Stopping Client Sessions and the Broker 70
Other Samples Available 71
SonicMQ Application Programming Guide 71
SonicMQ Deployment Guide 72
SonicMQ Administrative Programming Guide 73
Chapter 4: Next Steps 75
Glossary 79

Preface

About This Guide

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

The audience for this guide is developers, managers, and information architects who want an overview of messaging and SonicMQ. The reader is expected to be familiar with the concepts of distributed computing technology and the requirements they impose on development environments.

This guide consists of these sections:

- Chapter 1, "SonicMQ Architecture," introduces the SonicMQ messaging environment including some of its tools and major features.
- Chapter 2, "SonicMQ Messaging," presents Java Message Service (JMS) concepts and explains how these concepts are implemented and extended by SonicMQ.
- Chapter 3, "SonicMQ Sample Applications," provides step by step instructions for the sample programs, which demonstrate uses for SonicMQ.
- Chapter 4, "Next Steps," describes sources for learning more about SonicMQ after you run the samples, and tells how to uninstall SonicMQ.
- "Glossary" provides a list of commonly used words and phrases that describe SonicMQ and messaging functionality.

How to Use This Guide

Learn the concepts of the architecture and the essence of messaging in applications. Install the software then run the broker and some sample applications to get hands-on familiarity with SonicMQ.

Typographical Conventions

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

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

This manual uses the following syntax notation conventions:

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

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

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

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

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

SonicMQ Documentation

SonicMQ provides the following documentation:

Title and Format	***Description***
Getting Started with Progress SonicMQ (PDF)	An overview of the SonicMQ architecture and messaging concepts. Guides the user through some of the SonicMQ sample applications to demonstrate basic messaging features.
Progress SonicMQ Installation and Upgrade Guide (PDF)	The guide to maintenance of physical installations of SonicMQ software. Shows various ways to run setups and upgrades as well as the features to install on a system depending upon its intended use. Describes additional on site tasks such as defining additional components that use the resources of an installation, creating activation daemons, and encrypting local files, plus the use of characters and local troubleshooting tips.
Progress SonicMQ Deployment Guide (PDF)	Describes how to architect topologies of components in broker clusters, fault tolerance, and Dynamic Routing Architecture. Shows how to use the protocols and security options that make your deployment a resilient, efficient, controlled structure. Covers HTTP Direct, Sonic's technique that enables SonicMQ brokers to send and receive pure HTTP messages.
Progress SonicMQ Application Programming Guide (PDF)	Takes you through the Java sample applications to illustrate the design patterns they offer to 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.
SonicMQ API Reference ***(HTML)***	Online JavaDoc compilation of the exposed SonicMQ Java messaging client APIs.

Title and Format	*Description*
Progress SonicMQ Configuration and Management Guide (PDF)	Describes the container and broker configuration toolset in detail plus how to use the JNDI store for administered objects, and the certificate manager. Shows how to manage and monitor deployed components including their metrics and notifications.
Progress SonicMQ Administrative Programming Guide (PDF)	Provides information about moving development projects into test and production environments. Describes recommended build procedures, domain mappings, and reporting features.
Management Application API Reference (HTML)	Online JavaDoc compilation of the exposed SonicMQ management configuration and runtime APIs.
Metrics and Notifications API Reference (HTML)	Online JavaDoc of the exposed SonicMQ management monitoring APIs.
Progress SonicMQ Performance Tuning Guide (PDF)	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.
Sonic Event Monitor User's Guide (PDF)	Provides a logging framework to track, record, or redirect metrics and notifications that monitor and manage applications.

Worldwide Technical Support

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

When contacting Technical Support, please provide the following information:

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

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

Chapter 1 Introducing Progress SonicMQ

Introduction

Businesses are constantly searching for ways to improve efficiency and, therefore, their bottom line. Such efficiencies come from new or improved business processes that inevitably involve some form of integration. Examples of such integration are:

- Having several previously independent applications collaborate together through message passing
- Reliably deliver information from equipment in the field, for example: machinery within a process plant, sensors in medical equipment, or components in oil and gas pipelines
- Guaranteed delivery of data from remote business units into the corporate headquarters
- Reporting of business activity data to executive dashboards from distributed enterprise applications
- Secure and reliable transfer of data from business partners and customers into a trading exchange or trading portal solution

SonicMQ has been successfully deployed in mission-critical situations to solve all of the above problems—and many others. Its performant, reliable, secure, and scalable messaging has been the key to these successful projects.

At its core, messaging middleware is software that allows two entities to communicate by sending and receiving messages. In the same way that today's e-mail systems allow communication between two or more people, messaging allows communication among two or more applications, without requiring human intervention. These applications can reside independently on a wide variety of hardware devices.

Figure 1 shows a variety of devices communicating with each other using messaging concepts. A warehouse management system is running on a large server in the back office of a major supplier. This supplier offers access to customers via an on-line purchasing system. Customers enter their purchase requests, and appropriate messages are sent to the warehouse system to determine whether the order can be fulfilled, and if so, where from. At the same time, a store walker determines which store shelves need restocking and enters information into his application. That application then forwards the purchasing request directly to the warehouse, without requiring any user intervention.

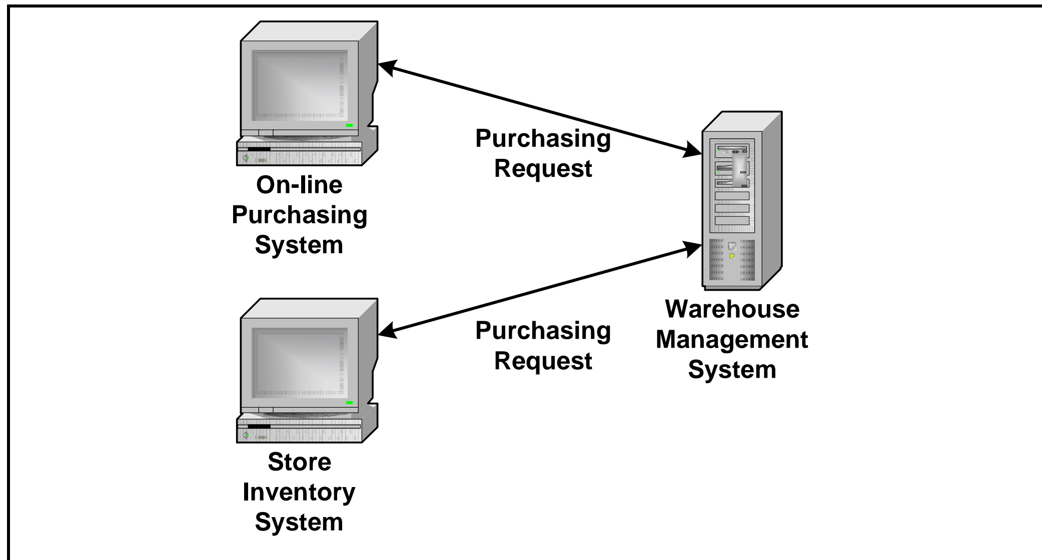


Figure 1. Warehouse Messaging Example

One of the most fundamental aspects of messaging is its asynchronous nature—the sender of the message does not have to wait for the recipient to receive the information. Going back to the e-mail analogy, once an e-mail has been sent, the sender does not wait for a reply—they are free to continue working while waiting for the recipient to respond. Sending applications are free to generate messages at an appropriate speed, handling peak periods as they occur, without having to wait for recipients to deal with the requests. There are instances, however, where synchronous communication is required (for example, awaiting the result of a complex mathematical calculation before making a decision). In these instances, it is essential that the messaging implementation that you choose can handle this type of communication, as well as the asynchronous form.

Messaging is being widely using today in support of business-to-business (B2B) and business-to-consumer (B2C) transactions, often occurring over the Internet.

What about RPC-Based Mechanisms?

Messaging is not the only way to communicate with other applications. Other technologies, such as CORBA, Java's RMI, and Microsoft's DCOM, all use a Remote Procedure Call (RPC) mechanism to transfer information among applications.

Figure 2 shows a typical RPC-based implementation. In this environment, clients and servers are tightly coupled to simulate a single process. All participants in the topology must be aware of each other, and all connections must be established before communication can take place. If you have to add an application client to this topology, all other application clients must be made aware of it, and you must establish a network connection to each of these application. When large numbers of application clients are involved, the topology becomes very complex.

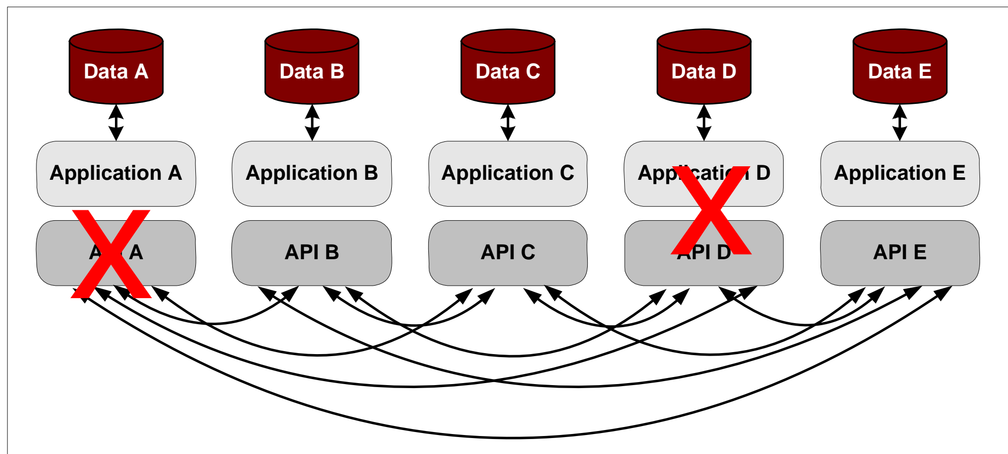


Figure 2. Typical RPC-based implementation

Contrast this topology with the loosely coupled topology required for the asynchronous messaging mechanism shown in Figure 3.

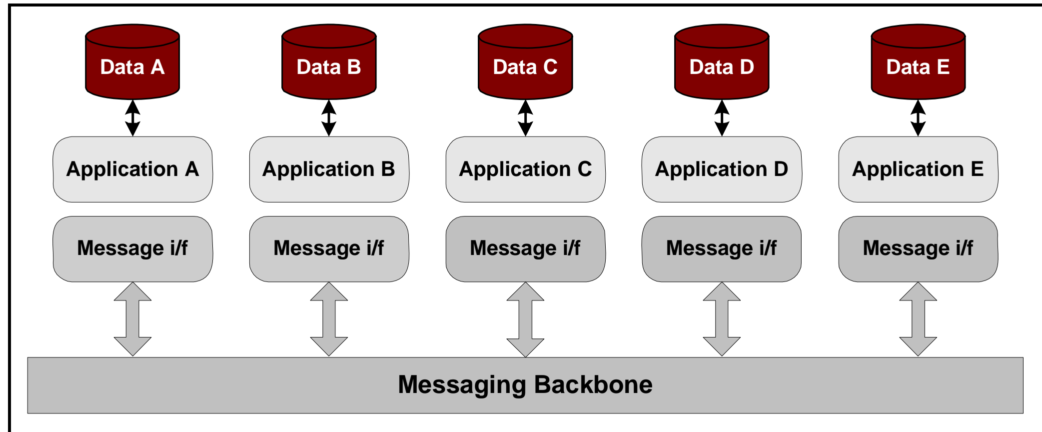


Figure 3. Application Clients Connected to a Messaging Server

In Figure 3, application clients are connected only to a messaging server. If any connection fails, the message server stores messages for that application client until the connection is restored. No other application client is affected by this failure. In a tightly coupled RPC-based topology, however, any failure causes a halt to the entire system until the connection is reestablished.

RPC-based mechanisms also cause sending applications to block, or wait for a task to complete, before continuing processing, forcing all processing to occur sequentially. The recipient of the information must complete its processing before the sending application can continue. In a loosely coupled asynchronous topology, however, after the message server has acknowledged a message sent to it, the sending application can continue processing, secure in the knowledge that the message server guarantees that the message is sent to its final destination reliably and securely.

What Problems Does Messaging Solve?

Messaging technologies have been with us since the 1970s, when they were used primarily to facilitate communication among back-office mainframe applications running on dedicated network connections. The messaging paradigm has risen in popularity over recent years due to changing business practices and technological advances.

Disparate System Integration

In the business world, mergers and acquisitions take place every day. More often than not, the merging organizations each have different internal systems that reflect their different business practices. To force one organization to change its internal systems in order to work with the application on the other side is a costly and sometimes disastrous endeavor. Decision makers often find that allowing these applications to coexist and communicate without disrupting everyday business processes is fundamental to ensuring a successful merger. Integration among applications is often required within an enterprise as well. Off-the-shelf applications are preferable to custom-built solutions in most enterprises. The wide variety of applications contributes to the disparity that large organizations face as they try to deal with many heterogeneous applications and platforms.

A low-cost, easy-to-implement, and flexible solution is required. Messaging technologies are often used, as they require little or no change to the underlying applications in order to guarantee communication among them. This decoupling of applications allows different business units to keep operating through the merger.

Exchanging Information With Business Partners

Conducting business successfully requires collaboration with a variety of business partners, both on the supply side and on the distribution side. All parties constantly exchange information, but not always reliably. For example:

- Price promotions are sent via faxes that are easily lost
- A telephone call is made to check on accounts payable, but the caller cannot reach the right party
- E-mail is used to place orders, but the numbers are transcribed incorrectly into the purchase order

You can use messaging technology to coordinate these communications and guarantee that the correct information is transmitted and received. The quality of service levels available with today's messaging systems ensures that the right information is exchanged among the appropriate parties, and that the appropriate security measures are in place.

Automating Common Business Functions

Streamlining today's business practices offers a measure of efficiency that directly affects the bottom line of a company's performance. Where possible, companies are looking at technologies that automate common business tasks or transactions that traditionally have required costly and slow user interaction.

Figure 4 illustrates a manufacturer alerting retailers about a wholesale price discount on one of its goods. This information is sent via a message to an application on the retailer's system. This application determines whether the offer is a good one (based on internally defined and maintained business rules), and if so, checks the inventory system and places an order for the required number of goods.

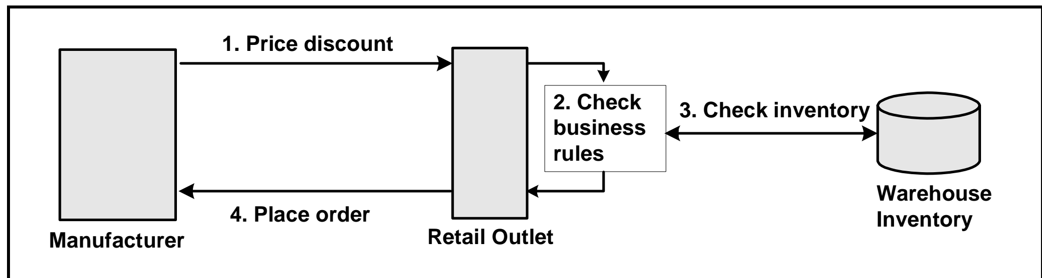


Figure 4. Messaging to an Application

Challenges of Building Next Generation Applications

The following sections discuss some challenges that exist in building the next generation of applications.

Portals and Trading Exchanges

There is no doubt that the Internet has significantly changed the way organizations do business. By removing geographic boundaries, the Internet has allowed businesses to exchange information and conduct transactions without requiring face-to-face interaction. Other technologies, such as XML, allow businesses to communicate in a standard way, giving them access to a wider range of potential business partners. These technologies, along with messaging, have given rise to the portal or trading exchange concept. These portals allow organizations that are interested in doing business with like-minded companies to seek out and conduct business with one another. By joining a trading exchange, companies can conduct business with all other organizations on that portal.

The trading exchange shown in Figure 5 describes an automotive portal. An automobile manufacturer (A) that requires a specialized component can put out a request to a trading exchange specific to the automotive industry. A small manufacturing company (X), which would not otherwise have access to this large organization, can bid to supply the required part.

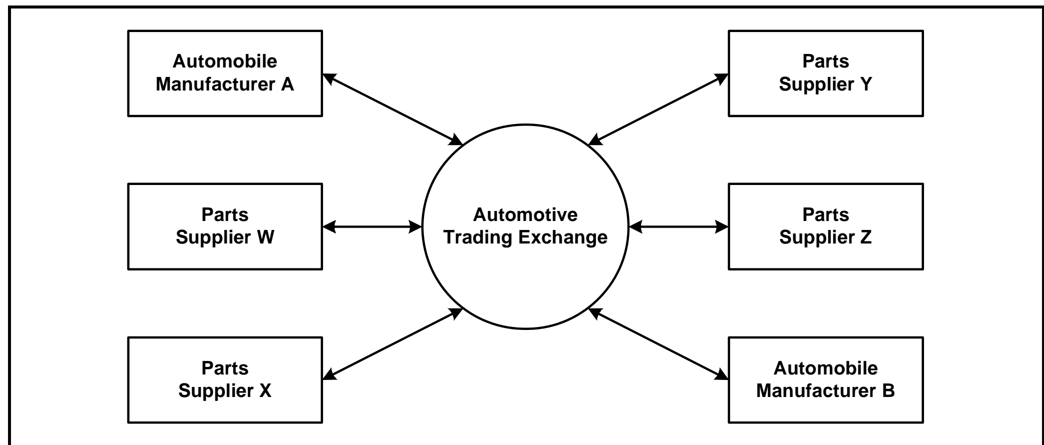


Figure 5. Trading Portal

Messaging systems are fundamental in trading exchanges. They ensure that information in the form of messages is sent reliably to a potentially massive number of recipients, while still providing a very high level of throughput and providing robust security.

Global Business Transactions

In today's global economy, organizations are thinking and deploying globally. Doing business in a geographically dispersed fashion presents its own sets of problems. How will the systems communicate? How can you keep costs down? How can you administer remote locations? The messaging model caters to geographically dispersed locations.

Figure 6 shows an example of a geographically dispersed deployment. An organization has regional headquarters in the Americas, Europe, and Asia/Pacific regions. Each of these headquarters services the local offices in their respective countries. The systems in these regional headquarters run independently of each other, but they want to exchange information regularly with the other headquarters. These deployments require a messaging system that can provide the high-level throughput required by the standalone infrastructures, while at the same time providing a mechanism for the headquarters to communicate with each other.

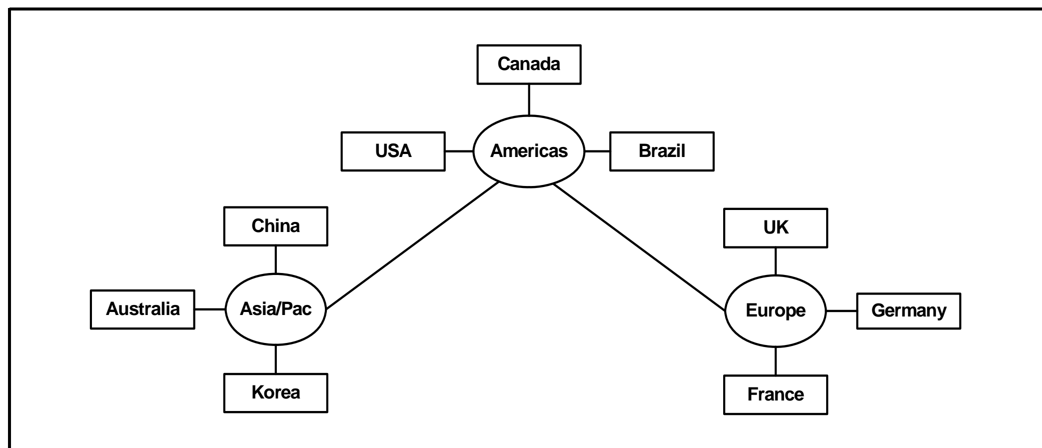


Figure 6. Geographically Dispersed Deployment

Progress SonicMQ: Software for Mission-Critical Messaging

Progress SonicMQ delivers everything a business needs to evaluate, design, deploy, and maintain all the most desirable features for high performance and high reliability messaging. For example:

- Certified JMS implementation that provides both queue Point-to-point messaging and Publish/Subscribe messaging
- Guaranteed message persistence over the Internet
- High scalability to support rapid topology expansion as well as intense changes in message volumes
- Industry-leading message security, encryption, authorization, and certificate management
- Built-in XML messaging and integration with XML parsers

SonicMQ's Dynamic Routing Architecture® (DRA) lets enterprises participate in global trading exchanges through communities of brokers. When trading domains come online, SonicMQ dynamically discovers their destinations and delivers messages between the servers on an optimized routing path.

The SonicMQ architecture sets a foundation for high-throughput messaging by robustly providing five architectural essentials: reliability, performance, scalability, availability, and security.

Guaranteed Reliability

Progress SonicMQ's guaranteed message delivery ensures that messages are never lost—even in the most challenging situations. This guaranteed delivery is supported by a message persistence technology designed for long term business success.

SonicMQ's local and distributed transaction support ensures data consistency throughout the enterprise.

Very High Performance

Benchmark results confirm SonicMQ's superior performance. Metrics show that a SonicMQ broker can support up to 2,000 concurrent connections with a 10MB per second throughput. SonicMQ maximizes server stability through effective use of flow control, which throttles client send rates to avoid message loss and ensures consistently high performance.

SonicMQ also implements an optimized persistence mechanism to maximize server performance for guaranteed message delivery. With its Concurrent Transacted Cache technology, SonicMQ utilizes both an in-memory cache and high-speed log files to increase throughput for short-duration persistent messages. Long-duration persistence (typically required for disconnected users) is supported through the embedded relational database or other JDBC-compliant databases.

Massive Scalability

SonicMQ's superior broker design ensures that high performance is achieved without sacrificing message reliability. Each broker supports thousands of persistent messages per second and can handle a vast number of connections and destinations.

SonicMQ's clustering capability allows the messaging workload to be shared among all the brokers, thereby improving performance and preventing any one broker from being overburdened.

In hundreds of large-scale, mission critical deployments, SonicMQ has demonstrated and proven its outstanding scalability.

High Availability

The SonicMQ messaging infrastructure is up seven days a week, 24 hours a day. SonicMQ's clustering allows multiple brokers to operate as a single routing node while managed from a centralized directory service. When clustering techniques are implemented, a client application can detect dropped connections, dictate load balancing algorithms, and get fault resilience support from the cluster.

End-to-end Security

Security is crucial in a global trading environment, and SonicMQ provides the following required security levels:

- SonicMQ brokers, clusters, and dynamic routing nodes manage access control to servers and destinations. A challenge-and-response protocol is implemented for user authentication.
- Pluggable authentication modules allow SonicMQ to be easily integrated into an enterprise's existing security environment.
- Transport protocols leverage Internet security mechanisms for messages and transport through multiple firewalls using flexible HTTP/HTTPS tunneling with forward and reverse proxy support.
- A message payload can be encrypted with an associated digest so that both message privacy and integrity are certifiable. The cipher used is pluggable giving choice over the strength of encryption. Even when the producer of a message is uncertain whether the broker destination demands payload encryption, the application can enable encryption on a per-message basis.
- Message security can use certificate-based mutual authentication through RSA's SSL implementation to get full server and client server security including certificate identity management and certificate generation. SonicMQ's SSL protocol implementation supports up to 256-bit encryption.

Management Efficiency

SonicMQ enables low total cost of management by providing these features:

- Manage entire deployments from a single Management Console
- Manage many brokers from a single Management Console
- Create an application configuration once, then push that configuration over many brokers from a single location
- Manage remote nodes from a single location
- Monitor application performance and messaging traffic remotely from a single Management Console for widely dispersed deployments

Chapter 2

SonicMQ Messaging

Messaging Concepts

This chapter provides an overview of JMS messaging concepts used in SonicMQ and an overview of some SonicMQ features that extend the JMS 1.1 specification. The JMS 1.1 specification is available on the Sun Developers Network at `java.sun.com/products/jms`.

Messaging Models

In JMS messaging, an application creates a connection, then establishes one or more sessions on the connection. Message producers send messages to destinations. Message consumers receive messages from destinations.

There are two messaging models that are used in JMS:

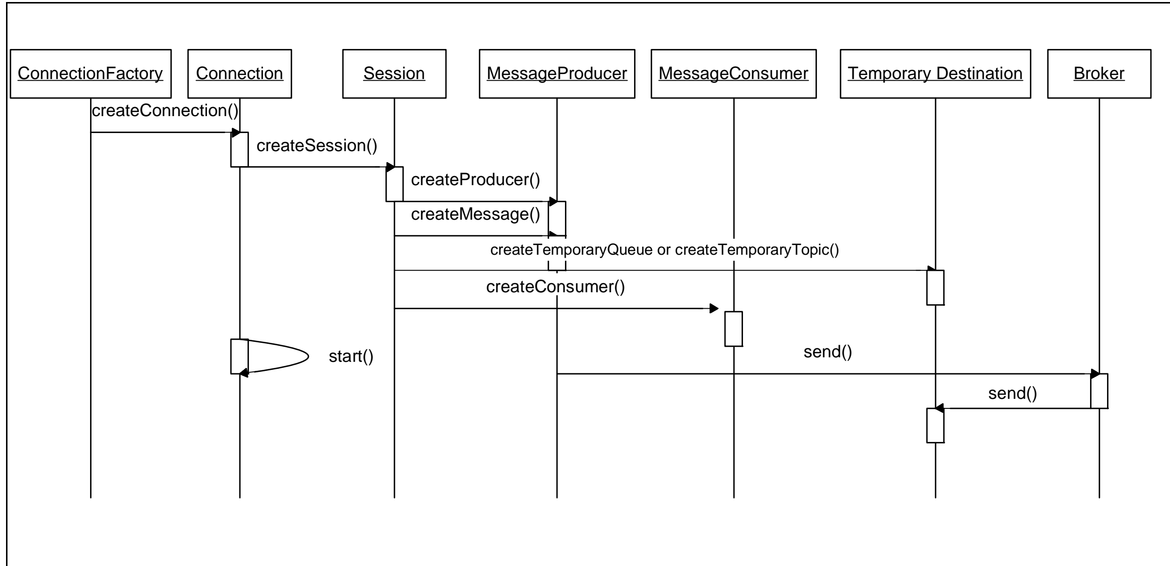
- **Point-to-point (PTP)** — One-to-one communication. A producer sends a message to a queue, where it is received by a single consumer, no matter how many consumers are listening to the queue.
- **Publish/Subscribe (Pub/Sub)** — One-to-many communication. A producer publishes a message to a topic, where all consumers subscribing to the topic receive the message.

These messaging models are substantially the same in terms of connection and session management, message structure, delivery mode options, and delivery methods. In both messaging models, a client application can be a message producer and a message consumer.

Before the JMS 1.1 specification, separate sessions were required for the PTP and Pub/Sub messaging models. JMS 1.1 provides a set of common interfaces that can be used for creating sessions that include both Point-to-point and Publish/Subscribe producers and consumers. This feature enables what the specification calls “unified domains.”

Unified Messaging

Unified messaging describes functionality that is common to both messaging models. The following sections describe the flow of objects as shown in this sequence diagram:



Connection Factories

Connection factories encapsulate connection information used by the application. They are either abstracted into administered stores where they are looked up or created by constructors in a client application. The preferred technique is to look up JMS administered `ConnectionFactory` objects stored in a JNDI store. SonicMQ provides an internal JNDI store that administrators can set up to store administered objects for lookup.

Connections

A SonicMQ application starts by accessing a connection factory to create a connection that binds the client to the broker. Connection factories are administered objects so the details of connection operations can be abstracted from the application.

As illustrated in the sequence diagram, a `ConnectionFactory` creates a `Connection` and then calls the `start` method. A `Session` is created in the context of a `Connection`. `MessageProducer` and `MessageConsumer` functions act on messages in the context of a `Session`.

Sessions

A client can create multiple sessions within a connection to the broker, each independently sending and receiving messages. Sessions execute in parallel meaning that multiple threads are active concurrently, thereby optimizing throughput. Figure 7 illustrates a client application where one connection is made through which one session is established.

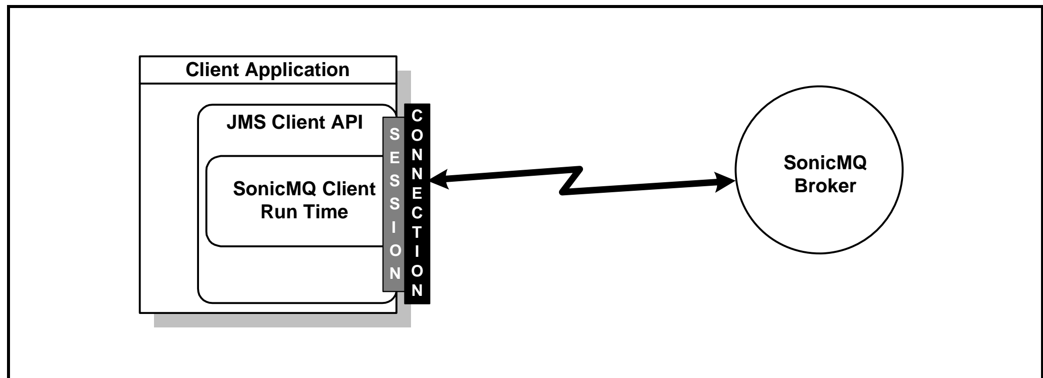


Figure 7. JMS Session on a Connection

Transacted Sessions: Commit or Rollback

In creating a session, you specify whether the session is ***transacted*** or ***nontransacted***. If a session is transacted, batch acknowledgement is sent when the `commit()` method is called. When a session is nontransacted, you set the acknowledgement mode.

Transaction processing significantly reduces the effort required to build applications by allowing applications to combine a group of one or more messages into a logical unit.

When a transaction **commits**, the message producer sends all the messages since the last transaction and the message consumer acknowledges the messages it received. If a transaction **rolls back**, the message producer drops any produced messages for the transaction and, for PTP, messages received by the consumer in the scope of the transaction are automatically recovered into their queues.

Note SonicMQ applications can also participate in **global transactions**: transactions that use XA Resources to extend beyond a single session. In global transactions, a transaction identifier is registered with a transaction manager that manages the various branches of the transaction as it is constructed and then controls the orderly preparation and commitment—or roll back—of the completed transaction.

Nontransacted Sessions: Set the Acknowledgement Mode

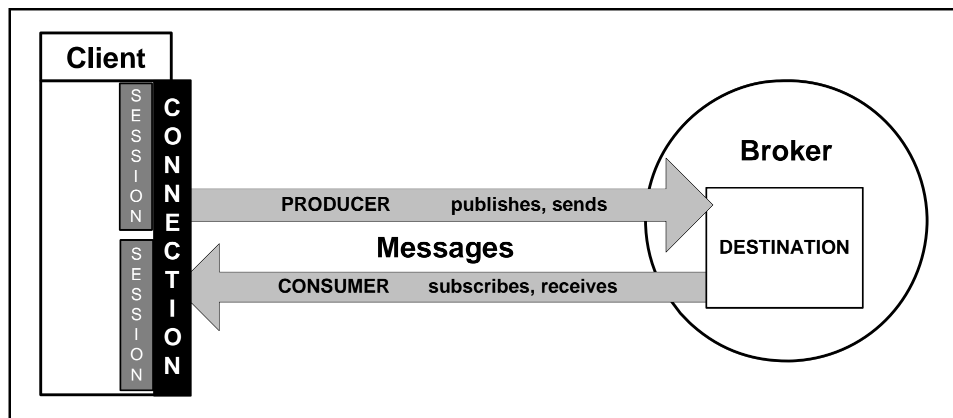
When an application creates a nontransacted session, the acknowledgement mode is set. The acknowledgement mode options are:

- **AUTO_ACKNOWLEDGE** — The client application session acknowledges receipt of a message when the session has successfully returned from a call to `receive`, or when the `MessageListener` (called to process messages) successfully returns a message to the client application.
- **CLIENT_ACKNOWLEDGE** — The client application calls a message's `acknowledge` method to acknowledge to the broker that all messages are not yet acknowledged.
- **DUPS_OK_ACKNOWLEDGE** — When the client session acknowledges the delivery of messages to consumers, the broker does not confirm the acknowledgement.
- **SINGLE_MESSAGE_ACKNOWLEDGE** — The client session explicitly acknowledges one specific message.

While acknowledgement sets standards for message delivery, there is no reply to the producer. The acknowledgment is always sent to, and consumed by, the broker.

Producers and Consumers

Within a connection and session, a client application produces or consumes messages. You could create multiple sessions and produce messages in one session and consume messages in another session. Creating one session for sending and another for receiving messages prevents bottlenecks and enhances performance by preventing one activity from blocking another activity. As illustrated, a client application can be both a producer and a consumer:



Producers

A message producer packages and encrypts the message body, sets the service level and protection for the outbound message, and then sends the message to its destination. If the delivery mode is `PERSISTENT`, the message will be placed in the broker's log or message store before starting delivery.

Message producers can set a time to live for each message. The calculated expiration time keeps a message in the queue until it expires. Expired messages are discarded.

Request/Reply

An application can use a request/reply mechanism to send a message and then wait for a response. The client application creates a temporary destination to receive the reply.

Quality of Service

Quality of Service refers to message delivery configuration that a producer specifies. Generally, Quality of Service affects the availability, reliability, scalability, and performance of message delivery. Loose Quality of Service uses a minimum number of features to maximize performance. A tighter Quality of Service uses multiple features, such as fault tolerant client connections, acknowledgements, and duplication elimination to maximize reliability, availability and transactional integrity.

For example, a loose Quality of Service might be used by a stock ticker application that wants to publish messages to a large number of subscribers to be consumed immediately. Since there are no security concerns and the information is time-sensitive, performance is at a premium. In contrast, a tight Quality of Service might be used for an online stock purchasing system, in which the company hosting the system wants to ensure transactional integrity, confidentiality, and reliability for all stock purchase transactions.

Quality of Service is supported by the following session options and message producer parameters:

- **Acknowledgement Mode** — The session determines whether the acknowledgment of communications between the client and the broker are controlled by the client, the broker, or are simply performed with reasonable efforts. Adding acknowledgements to messaging creates an additional demand on a client application, but also assures greater reliability.
- **Message Expiration** — Messages can be sent with a specific life span to make sure clients do not receive out-of-date information. When a message expires, it is dropped from the queue or from the messages stored for disconnected subscriptions.

- **Delivery Mode** — A persistent message is stored in the broker's logs and repository for later delivery to potentially disconnected users. This action provides a higher Quality of Service yet results in a corresponding decrease in performance. A persistent message survives system disconnection, or unexpected restarts. Persistence is maintained as a message is routed across brokers.
- **Guaranteed Persistence** — A message can be sent to a system **dead message queue** (DMQ) if it expires or is undeliverable. On the DMQ, a message never expires and can only be discarded by explicit administrative action.
- **Priority** — Messages can be sent with a priority value that encourages the broker to position that message ahead of other messages in the same queue or topic for delivery.
- **Redelivery** — The broker can make repeated attempts to redeliver messages to each client that has not acknowledged receipt.

Consumers

A message consumer binds to a destination on a broker to receive messages. When messages are received, a message consumer acknowledges receipt to the broker according to the parameters set on the session.

Synchronous Message Consumption

A synchronous consumer uses a pull technique to receive messages from a destination. There are three variations of the `receive` method, one which never blocks, one which blocks until it times out, and one which blocks indefinitely (meaning that the connection could be blocked indefinitely too).

Asynchronous Message Consumption

An asynchronous consumer registers a message listener on a destination. As messages arrive, the application calls the listener's `onMessage` method. When the connection is broken, messages that are guaranteed to be delivered wait on the broker for reconnection.

Message Selectors

Message consumers can filter the messages they receive. When a client requests a session to create a consumer on a topic or queue, a message selector string in SQL-92 syntax can be used to indicate qualifications on messages. For example:

```
"Priority > 7 AND Form = 'Bid' AND Amount is NOT NULL".
```

The Message

A message is comprised of:

- **Header** — Contains name-value pairs used by applications to identify and route messages, such as a time stamp, the expiration time, the priority, the delivery mode, the destination, an identifier, and settings used for correlation, redelivery, reply destination, and an arbitrary type (not the JMS message type.)
- **Properties** — Can be any of several data types: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, or `String`. Custom-defined properties provide name-value pairs that can be named, typed, populated, sent, and then coerced by the consumer into other acceptable data types. SonicMQ defines a few properties that are used to manage undeliverable messages, set per-message encryption, and specify message types.
- **Body** — Set of bytes interpreted as the designated message type. The message body is optional; you can send a message where the header and property values contain all intended information. SonicMQ provides the six message types defined by JMS—`Message`, `TextMessage`, `ObjectMessage`, `StreamMessage`, `MapMessage`, and `BytesMessage`—and extends two of the JMS message types:
  - The `TextMessage` is extended to implement the `XMLMessage` type.
  - The `Message` is extended to implement the `MultipartMessage`.

Destination: Topic or Queue?

The unification of JMS messaging models, introduced in the JMS 1.1 specification, provides generic connection, session, message, message producer, and message consumer interfaces. The advantage is that you can have queue-based messages and topic-based messages in a single transaction.

You can declare messaging model specific methods at any point, but you are then bound to that messaging model. For example, you can use a generic `ConnectionFactory` to create a `QueueConnection`. Then, you would have to create a `QueueSession`, `QueueSender`, and `QueueReceiver`.

Eventually you have to declare what set of behaviors to apply to a message. This occurs when you specify the destination. A destination is either a ***topic*** or a ***queue***. You can send to a statically defined queue or publish to a dynamically defined topic, both with the same name. The broker considers these different types of destinations—one is a queue and one is topic. They have different authorization rules and require different handling. The next two sections describe and differentiate between each of the messaging models.

Publish and Subscribe Messaging: Broadcast the Message

In the Publish and Subscribe (Pub/Sub) messaging model, one client application can send a message to many other client applications. In the Pub/Sub model, a message producer is a ***publisher*** and a consumer is a ***subscriber***.

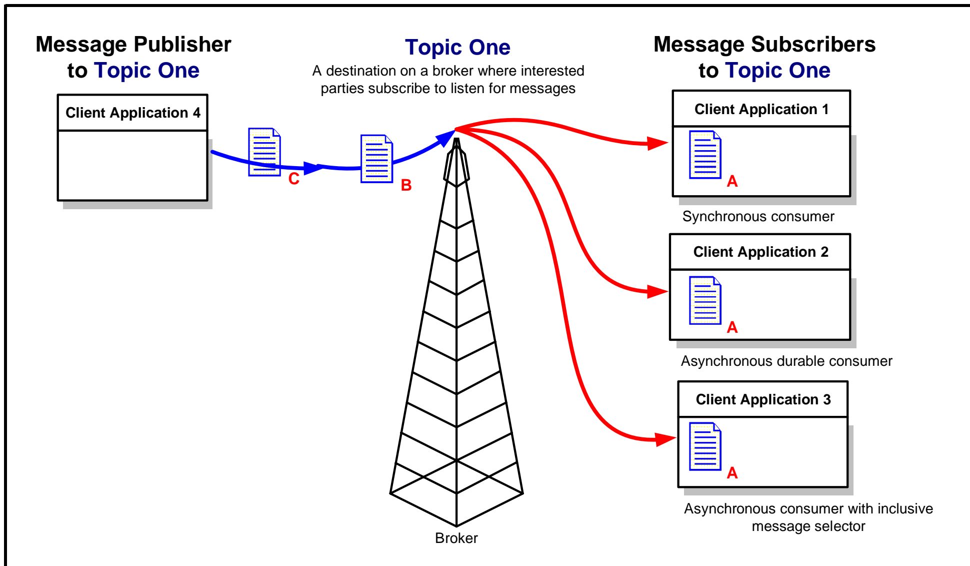


Figure 8. Concept of Publish and Subscribe Messaging Topics

Figure 8 shows three subscribers who have each received message **A** and are about to receive message **B**, and then message **C**:

- **Client application 1**, a synchronous subscriber, waits for a message, for a specified time or forever, and then blocks to receive again after processing a message.
- **Client application 2**, a durable subscriber specified an interest in receiving messages from the broker on the selected topic, even when disconnected. Messages are saved for durable subscribers, although a saved message can expire while waiting for the durable subscriber to reconnect.
- **Client application 3**, an asynchronous subscriber has set up a message listener. When a message arrives, the `onMessage` method in the client is called. Although it is not shown in Figure 8, this subscriber has a message selector. The subscriber provided a string in SQL syntax as a parameter when the subscription was created. If the selection criteria are not met, the subscriber does not take delivery of that message.

Figure 9 illustrates the distribution of message delivery. The shapes on the publisher lines indicate messages published to specific topics over time. The crossovers within the broker represent the subscriptions that could have wildcard patterns so a single subscription delivers messages from many topics. The shapes on the subscriber lines indicate that everyone subscribed to a topic receives the same message at the same time. The exception is the durable subscription that, when not active, can receive its messages later, provided the messages do not expire.

Figure 9. Publishing Messages to Topics for Subscribers

Durable Subscribers

There are several special techniques for registering a durable interest and to distribute the load of subscribed messages but they all involve the notion of a subscription: if your consumer (or registered interest) is there when the message arrives on the topic, you and everyone else there gets that message. If you missed it (and did not register a durable interest), it is gone.

Any messages that are not acknowledged by durable subscribers when they are not connected to the broker, are preserved by the broker and redelivered when the durable subscriber reconnects. When a durable subscriber reconnects, messages are delivered in the correct order, although some messages might have expired. If a durable subscriber does not acknowledge delivery of a message, the message is retained until it expires.

Point-to-point Messaging: Each Message Has One Consumer

The Point-to-point (PTP) messaging model ensures that a message is delivered only once to a single consumer. Figure 10 shows three message producers sending messages to three different queues. Queues are defined with a maximum size and a threshold that indicates at what point to persist messages on the queue in the data store. If messages are taken off the queue at the same rate they are placed on the queue, then no messages are persisted.

In the diagram, **Producer 2** sends a message to **QueueB**, which has three registered listeners. Because only one receiver will receive this message, the broker decides to let **Consumer B1** receive the message. The other consumers do not receive that message.

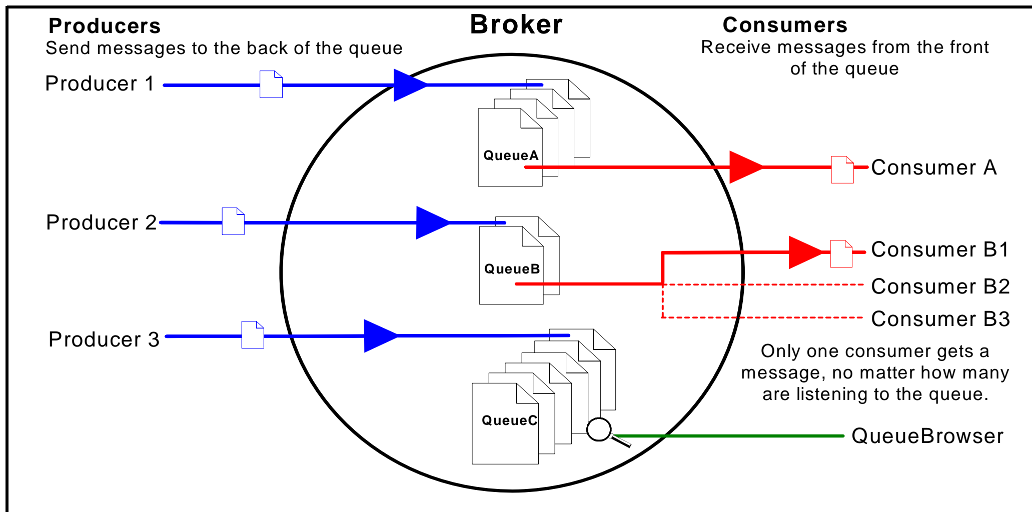


Figure 10. Sending Messages to Queues for Receivers

Each message producer sends new messages to a queue, a destination on the broker. The broker, unless advised that there is a request for priority treatment, places new messages at the back of the queue. **Consumer A** takes the frontmost message off **Queue A**.

On **Queue B**, multiple consumers are listening to the queue, but only one consumer receives the message, **Consumer B1.** The **QueueBrowser** is browsing the queue without taking messages off the queue.

Queues are useful for load-balancing when many diverse systems share processing operations because additional receivers can be used to keep queue processing current. For example, applications for trading activities, credit card charges, online shopping carts, auctions, reservations, and ticketing often use queues.

Characteristics of the Point-to-point messaging model include:

- Point-to-point messaging queues are static. They are created by an administrator through the Sonic Management Console or the Management APIs.
- The first message received by the broker is the first message delivered. This “First In, First Out” (FIFO) technique makes the second through **n**th messages endure until that first message is consumed. (Note that mixed priority settings on messages affect FIFO.)
- Even when no clients specify an interest in receiving messages from a queue, messages wait for a consumer until the message expires.
- When a message’s delivery mode is set to `PERSISTENT`, the message is stored so that even a broker shutdown will not put the message at risk.
- There is only one message consumer for a given message. Many consumers can listen or receive on a queue, but only one takes delivery of a specific message.
- When the message is acknowledged as delivered by the consumer, it is removed from the queue permanently. No one else sees it and no one else receives it.

Figure 11 illustrates other features in SonicMQ Point-to-point messaging.

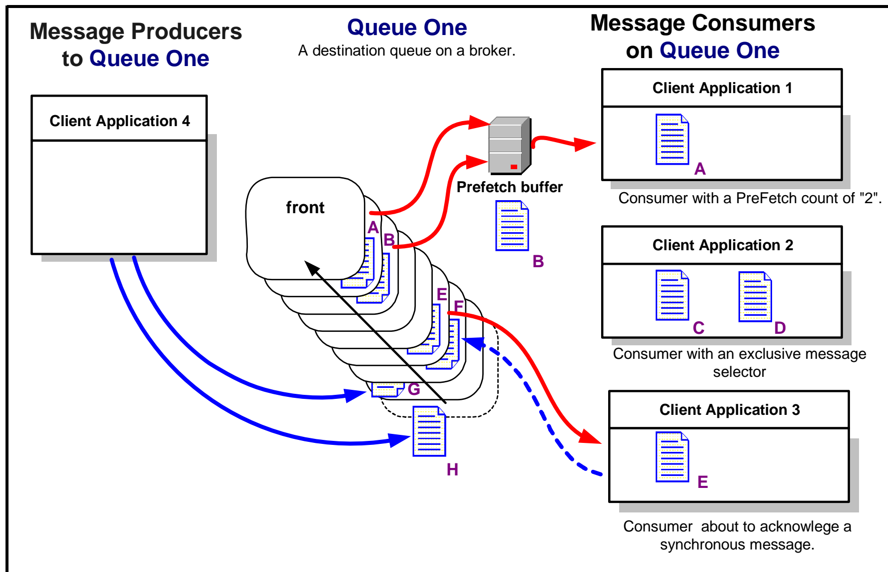


Figure 11. Features of Point-to-point messaging

The messages in this illustration are being received from the queue by three consumers:

- **Client Application 1**,with a prefetch count of **2,** took messages **A** and **B** off the queue. Message **B** is held in the prefetch buffer while message **A** is processed. When message **B** enters processing, the threshold trigger causes the consumer to draw off two more messages. The broker keeps track of unacknowledged messages and, if acknowledgement is not received before the session closes, the unacknowledged messages are reinstated in the queue.
- **Client Application 2**, consuming with a message selector, reviews qualified messages on the queue to filter the messages that it wants to process. In this example, the consumer selected and acknowledged messages **C** and **D**. Assuming message **F** does not meet its criteria, this consumer perceives a momentarily empty queue.
- **Client Application 3**, a synchronous consumer, receives a message, acknowledges its receipt, processes the message, and then waits to receive another message. The queue state indicates that message **E** was the frontmost message because messages **A** and **B**, while still in the queue, are awaiting acknowledgement from another consumer.
- **Client Application 4**, a message producer, has sent message **G** to the queue and is in the process of sending message **H**.

SonicMQ Messaging Clients

The behaviors described in this chapter apply to the SonicMQ messaging clients. These are distinguished form management clients, ones that programmatically perform configuration and runtime tasks. There are several SonicMQ clients to consider:

- **Java Client** — The fundamental client is the Java client. All functionality described in this document and all the general SonicMQ documentation refer to the Java client.

  The Sonic Java client supports the notion of ***bridges***, techniques that connect SonicMQ brokers to other messaging brands and other transports such as FTP and mail. Bridge software is supported on the third-party end by the appropriate third-party client and, on the side that faces the SonicMQ broker, the SonicMQ Java client.
- **HTTP Direct Clients** — SonicMQ provides special protocol handlers for HTTP connections so that inbound HTTP messages can be received and acknowledged as though the SonicMQ broker were an HTTP Web server. The HTTP messages are transformed to pure JMS messages using a combination of defined properties and properties in the message. There are also techniques to send messages to HTTP servers using the PUSH method.
- **C/C++/COM Clients** — SonicMQ provides its client software for pure ANSI C, C++, and COM development. These are substantially complete JMS clients and support much of the functionality of the Java clients, though there are differences. The differences are described in the guide for each client.

SonicMQ Features for Developing Client Applications

SonicMQ's extensions of the JMS specification make it easier to deploy and manage sophisticated message systems.

Features for Publishers and Subscribers

SonicMQ provides many extensions to the JMSPublish/Subscribe messaging model to provide greater functionality and flexibility. Some key features for subscribers follow.

Remote Publishing

Remote publishing provides a way to use the Dynamic Routing Architecture to dynamically pass a published message through a connected broker to a remote broker where it is delivered to connected subscribers to the target topic on that broker.

DISCARDABLE Delivery Mode

Additionally, Pub/Sub offers a discardable delivery mode so that high-volume publishers can provide relief to slow subscribers that fall behind the incoming message flow. In this mode, messages are discarded when it is reasonable to do so.

Subscription to Topic Hierarchies

SonicMQ extends the JMS standard topic naming mechanism with topic hierarchies (also referred to as **hierarchical name spaces**) specified with a dot-delimited string, such as `Orders.Euro.Gov`. The effort is minimal for the publisher, yet there are significant administrative and security advantages to the client subscription. The subscribers to the topic node can, if granted permission to do so, subscribe to all orders (`Orders.#`), only Euro orders (`Orders.Euro.*`) or any other combination with a few template characters. Subscribers to the root topic (`""`) get all messages by using (`#`).

Shared Subscriptions

SonicMQ allows groups of subscribers to share messages from a shared subscription. By simply using a group name offset by special characters, a subscriber to **TopicA** can instead subscribe to **[[myShareGroup]]TopicA**. When other subscribers subscribe in that same way, they take turns receiving the latest message on the one subscription. This provides a simple form of load balancing to subscribers.

Global Subscriptions

Brokers can act as subscribers, enabling subscribers to a topic on one broker to receive messages that the broker received from its subscription to that topic on another broker.

Flow Control

SonicMQ allows you to manage the flow of messages between broker and client. Managing message flow is important when clients produce more messages than the clients consuming them can process. System resources for storing messages, such as destination message buffers or maximum queue size, can be exceeded if too many messages must be stored. SonicMQ uses flow control to stop accepting messages from producers until space becomes available.

Flow to Disk

When flow control demands that publishers wait until message space is available, the flow-to-disk feature provides a way for brokers to accept messages even though flow control has come into effect. Publishers are able to keep publishing until the maximum flow-to-disk memory on the broker is exceeded.

Option to Preserve Undelivered Messages

Through message properties, a message can indicate an interest in being retained if it expires without being delivered or becomes undeliverable. Every broker maintains a dead message queue. Messages transferred there are set to never expire and retain their original destination. Applications can pick up these messages and reattempt delivery or route them to personnel or applications for special handling.

Dynamic Routing Architecture

Dynamic routing provides techniques for secure interbroker communications that are designed to transcend domain and enterprise boundaries. When brokers define routing to other nodes, messages sent to a routing defined on a broker are forwarded to the stated destination on the remote broker. When defining a message destination, the routing defined on the connected broker is part of the destination name. See "Routing Nodes and the Dynamic Routing Architecture" on page 17 for an illustration of dynamic routing.

Security

SonicMQ provides security options that can be mixed and matched to provide the best balance of secure message transfer and performance in applications. For example:

- Secure protocols such as SSL and HTTPS can be used in one connection to a broker while another connection could use unsecure protocols. Interbroker communication and dynamic routings can use channel encryption.
- Authentication can use the client side login SPI, external authentication, or built-in authentication techniques.
- Authorization on destinations and routing is based on user groups or destination-specific settings. These can be defined with wildcards to provide ranges of permissions when hierarchical destination name spaces are used.
- Quality of Protection sets the level of protection and encryption required for a message. Settings on the destination at the broker define whether to trap altered messages. The application can choose to set the preferred level on messages sent from the producer to the broker.

Duplicate Message Detection

While single delivery of a message can be guaranteed, the message producer in a transacted session could be instructed to send another copy of a message already sent. To eliminate messages that are resent from being processed, the producer can add an identifier (such as an order number or authorization number) to a message as an audit tracker that is stored for a defined time span in a broker persistent storage mechanism reserved for duplicate detection. Before the broker commits a transaction, it attempts to insert the transaction’s identifier into the persistent storage mechanism. When an identifier is found to already exist in the persistent storage mechanism, the broker rolls back the transaction as a duplicate that has been processed successfully.

Recoverable File Channels

SonicMQ provides support for large message transfers through 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. A transfer breaks up a message into fragments and then monitors the progress of the transfers from peer to peer. If a transfer is interrupted, it can resume from the last checkpoint.

Client Persistence

SonicMQ allows a 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.

This chapter describes basic JMS messaging and some of the features in SonicMQ. In the next chapter, you install SonicMQ and experience SonicMQ messaging.

Chapter 3

SonicMQ Sample Applications

Getting Started With Selected Samples

The samples in this Getting Started Guide show fundamental SonicMQ messaging functions without requiring any elaborate setup, security, or multiple broker topologies.

Publish and Subscribe (TopicPubSub Folder)

In the Publish and Subscribe messaging model, several sample applications run in multiple console windows to demonstrate how messages are produced and consumed through topics. These applications include:

- **Chat Application** — When sessions are running `Chat`, a message entered in one session is displayed in all windows.
- **Message Monitor Application** — Uses a Java window to monitor the messages in the entire topic name space.
- **Durable Chat Application** — When sessions are running `DurableChat`, messages look similar to `Chat` messages, but if one of the sessions is interrupted, messages are retained for it by the broker.
- **Shared Subscriptions Application** — By declaring a group name, several subscribers take turns receiving the latest published message.
- **Selector Chat Application** — Message delivery is constrained by the defined message selection criteria.
- **Hierarchical Chat Application** — Demonstrates the advantages of using SonicMQ topic trees over message selectors.
- **XML Messages Applications** — `XMLDOMChat` messages and `XMLSAXChat` messages are translated into XML for publication and then interpreted when received by a subscriber.

Queue Point-to-point (QueuePTP Folder)

The Point-to-point samples are similar to the Pub/Sub samples. These samples show some of the common behavior across messaging models as well as the differences between the messaging models. for example:

- **Talk Application** — When sessions are running `Talk`, a message entered in one session is displayed in one other consumer window.
- **Queue Monitor** — Uses a Java window to review the messages waiting in a specified queue. While sessions are running `Talk` without queue consumers, the messages that are waiting in the queue are browsed.
- **ReliableTalk Application** — A connection is monitored for exceptions, and the connection is re-established after it drops.
- **Transacted Session** — A set of entries is buffered until a command indicates that the set of messages can be either sent (committed) or ignored (rolled back). The buffer then flushes. This sample uses two sessions to separate message producers and consumers.
- **Request and Reply** — A `Replier` is set up to consume a text message, convert it to all uppercase characters, and then send it to the temporary queue where the `Requestor` said it will wait for reply.

At the end of this chapter, other samples packaged with SonicMQ are outlined so you can locate them in the other guides in the SonicMQ documentation set. Those books provide context for the more advanced samples and describe the supporting tasks that make the sample function.

Many of the other guides in the SonicMQ documentation set also have examples. These examples describe how to setup working models of channel encryption, interbroker functions, Dynamic Routing, fault tolerant clients, replicated brokers, and fault tolerant management components.

Installing SonicMQ to Run the Sample Applications

When you are ready to run the sample applications, a **Typical** installation of SonicMQ assures that the instructions in this guide are consistent with your software settings.

Under Windows, the preferred Java software will be installed by the installer. Browse to `sonicsoftware.com/support` to access information about supported platforms and Java software versions.

See the *Progress SonicMQ Installation and Upgrade Guide* for information about installations options, features, and configurations.

◆ To install SonicMQ for evaluation of the sample applications:

1. Access a SonicMQ V7.5 license key either by registering at `sonicsoftware.com` or locating your assigned key in your distribution package.
2. On the system where you want to run the sample applications, insert the Sonic Software media or unpack the download file.
3. Locate the `setup` script:
   - Under Windows, `setup.bat`
   - Under UNIX or Linux, `setup.sh`
4. Launch the `setup` script:
   - Under Windows Explorer, double-click on `setup.bat`
   - Under UNIX or Linux, enter `./setup.sh`

   The SonicMQ Installer wizard opens.
5. Follow the wizard prompts for a **Typical** installation of **SonicMQ**.
6. When requested, enter your SonicMQ V7.5 license key.
7. You can specify a preferred installation location if you want, but otherwise accept all the default settings and complete the installation.

Important The default installation of the **Typical** set of features provides two brokers on one system. By default, the management broker accepts connections on port **2506** while the messaging broker accepts connections on a different port, **2507**. If you performed a **Custom** install of the **Domain Manager** on one system, and a **Messaging Broker** on another system, each would be assigned port **2506** on its system.

The samples default to using port **2506** so that, in either circumstance, the broker **localhost** and the port **2506** accept client connections.

After the Installation Is Complete

When the installation has completed successfully, the software is ready to run. You do not have to reboot the machine. The default settings installed the following features.

Feature Group	*Features*
Messaging Infrastructure	• **Domain Manager** — The framework components that enable a complete domain are installed. The domain is identified as **Domain1**. Includes a broker, a container, and the JMS client. • **Messaging Broker and Container** — The persistent storage mechanism used for messaging is setup and initialized. Client messaging functionality for both management communications and messaging is set up. This broker is assigned the name of the host system and it is listening for TCP connections on port 2506. Includes a container, and the JMS client. When security is not enabled, it is easier to run the sample applications. • **Container** — A container with the name of the host system is installed to host services and other configuration objects. Requires: ■ **JMS Client** — The Java JMS client libraries plus libraries for client persistence, recoverable file channels, and SSL.
Client Software and Samples	• **JMS Client** — The Java JMS client libraries plus libraries for client persistence, recoverable file channels, and SSL. • **Sample Applications** — The complete set of sample applications is installed in the `/samples` directory. Some of these require additional set up of features and are discussed at the end of this chapter.

Management Tools	• **Admin Tools** — The Sonic Management Console. This requires:
	  ■ **JMS Client** — The Java JMS client libraries plus libraries for client persistence, recoverable file channels, and SSL.
	  ■ **JMS Test Client** — The visual client tool to test text and XML messaging behaviors.

Other	• **Documentation** — Complete books and API reference for this release of SonicMQ.

Starting the Broker

The samples in this book focus on client applications interacting with a broker. First, you start the container that hosts the broker, and then open console windows to run some precompiled sample applications.

◆ To start the SonicMQ broker:

- On Windows, choose:
  Start > Programs > Progress Sonic > SonicMQ 7.5 > SonicMQ DomainManager.
- On UNIX or Linux, open a console window, `cd` into the installation directory, and then enter `../bin/startcontainer.sh`

The SonicMQ DomainManager console opens and displays its information as the broker starts up. The broker is ready when it displays:

(info) Management connection (re)established

You can minimize the window. Closing the window, however, will stop the broker.

Preparing to Run the SonicMQ Samples

After you successfully install and start SonicMQ, you can run the sample applications. The SonicMQ samples are sample applications that demonstrate typical JMS messaging features. Source code is provided for each of the sample applications in the directories where you run the samples.The samples in this book are grouped into sets of samples for each messaging model. Publish and Subscribe samples are presented first followed by the Point-to-point samples.

If you installed SonicMQ:

- **On Windows** — The samples in this chapter work as described.
- **With security enabled** — Usernames in the samples provide information to keep track of multiple application instances. When security is not enabled, the **`-u`** parameter accepts whatever name you enter. But when security is enabled, you must either define the users or use the default username **`Administrator`** throughout. Then, once users are established, you must add the password (**`-p`**) parameter and the appropriate password on every command line. For example:

  ```
  ..\..\SonicMQ Chat -u myChatUser -p myChatPassword
  ```

  User names cannot contain the characters, asterisk (`*`), pound (`#`), dollar sign (`$`), or backslash (\).

- **On a UNIX or Linux platform** — The samples in this chapter work as described, but you must use the shell scripts (`*.sh`) rather than the batch files (`*.bat`) and substitute a forward slash (/) for any instance of a backslash (\); for example:

```
../../SonicMQ.sh Chat -u Market_Maker
```

- **On a remote system or different port** — 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
```

Client Console Windows

Each sample application instance is designed to run in its own console window with the current path in the selected sample directory, for example,

MQ7.5_install_root\samples\TopicPubSub\Chat

When you stop a sample application, you can reuse the console window.

Windows Shortcut

From the Windows **Start** menu you can quickly open standard **Command Prompt** console windows at the root of the SonicMQ installation.

Choose **Start > Programs > Progress Sonic > SonicMQ 7.5 > Command Prompt**

Using the Sample Scripts

Each of the sample class files is located in a folder of the same name within its messaging model. Accompanying each `.class` file are its Java source file and a `readme.txt` file. For example, the Publish and Subscribe sample `Chat` is:

MQ7.5_install_root\samples\TopicPubSub\Chat\Chat.class

A universal script handler is installed at the `Samples` directory level. This script, **SonicMQ**, sets the local Java **\bin** path and then invokes the executable with its parameters, the **CLASSPATH**, and a list of variables. The script runs the sample applications. Standard invocation of the script from a sample folder is, in most cases, two levels down:

..\..\SonicMQ *application parameter1_name parameter1_value ...*

Note Consider all text to be case-sensitive. While there may be some platforms and names where case is not distinguished, it is good practice to always use case consistently, and, when specified, enter the case provided.

TIP You can use the SonicMQ JMS Test Client to view message header field and property values. But be aware that while using the JMS Test Client as a subscriber, it is just another subscriber but in the PTP samples the JMS Test Client is a consumer. Because PTP delivers a message to only one consumer, this might distort the expected results in other console windows.

Topic Publish and Subscribe Samples

The first samples to explore use the Publish and Subscribe messaging model. These samples progress:

- From demonstrating basic Pub/Sub messaging behavior,
- To showing a GUI window that tracks the messages,
- To showing how a subscription can be active when the application is not connected,
- To showing how several subscribers can share one subscription,
- To showing how to screen messages,
- To showing how to subscribe to sets of topic names,
- To showing how XML messages can be handled in DOM or SAX.

Chat Application

The `Chat` application shows the most basic Publish and Subscribe activity: whenever a message is published to a given topic, all subscribers to that topic receive that message.

In the `Chat` session diagram shown in Figure 12, each of the groups of steps describes a process in the `Chat` application running on the client while it communicates with the broker. The processes that involve both the client and the broker are in process blocks. The extent of the active connection is gray.

Client
Broker

Create a Connection and Sessions
- **create connection factory** for broker and application
- **create connection** for authorized user
- **create sessions** (nontransacted, automatic acknowledgement):
  - **Producer session** for input and sending
  - **Consumer session** for message listener and receiving

Create a Producer and a Consumer
- create message producer
- create message consumer
- create topic
- start the MessageListener

Start the Connection
- connect.start()

Produce Message(s)
- create a message
- when message is ready, set header and property values
- publish the message

if (stdin > null)
msg=username+":"+stdin
Message
publisher.send (msg)
User Entity
as Producer

Receive message for topic.
-THEN-
Publish message to subscribers to topic.

Listen and Receive Message(s)
- onMessage event, get the message
- read the message header, properties, and body type

string = textMessage.getText()
getText
Message
onMessage (msg)

Consume Message(s)
- Do action (message)

system.out.println (string)
User Entity
as Consumer

Close the Connection
- connect.close()

Figure 12. Diagram of the Chat Application Functions

◆ To start Chat:

1. Open a console window to the `Chat` folder.

TIP If you performed a default Windows installation, you can do either of the following:

- Open a console window on the `C:` drive then enter:
  `cd ProgramFiles\SonicSoftware\SonicMQ\samples\TopicPubSub\Chat`
- Choose **Start > Programs > Progress Sonic > SonicMQ 7.5 > Command Prompt** then enter:
  `cd samples\TopicPubSub\Chat`

2. Enter `..\..\SonicMQ Chat -u Market_Maker`
3. Open another console window to the `Chat` folder.
4. Enter `..\..\SonicMQ Chat -u OTC_Ticker`

Note The `-u` parameter specifies the user name. The samples all require that you enter a user name but do not require that you enter a password unless security was enabled. Usernames cannot use the characters asterisk (`*`), pound (`#`), dollar sign (`$`), slash (`/`)or backslash (\).

◆ To run Chat:

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

After running the `Chat` sample, consider that message services enable inter-application communications without synchronous threads. For example, stock quotes from markets could be streaming to subscribers in other markets who display them on scrolling displays. If they miss some of the messages, they just pick up the latest whenever they reconnect to the broker. The message volume could be huge, and out-of-date messages are not useful, so no messages are retained and no messages are guaranteed to be delivered.

Leave the `Chat` sessions running and proceed to the next sample where you will open a Java window to monitor the message traffic.

Message Monitor

`MessageMonitor` is an example of a supervisory application with a graphical interface. The application listens for any message topic activity by subscribing to all topics in the topic samples hierarchy, and then displays each message in its window.

◆ To run MessageMonitor:

1. Open a console window to the location:
   MQ7.5_install_root\samples\TopicPubSub**\MessageMonitor**.
2. Enter:
   ..\..\SonicMQ MessageMonitor
   The console window indicates that it has subscribed to **jms.samples.#**.

   The `MessageMonitor` window opens.
3. In one of the **Chat** windows, type text and then press **Enter**. The text is displayed in both **Chat** windows and the **MessageMonitor** window.
4. In the other **Chat** window, type text and then press **Enter**. The text is displayed in both **Chat** windows and the **MessageMonitor**.

As shown in Figure 13, several messages appear in the `MessageMonitor` window that were published to the topic, `jms.samples.chat`.

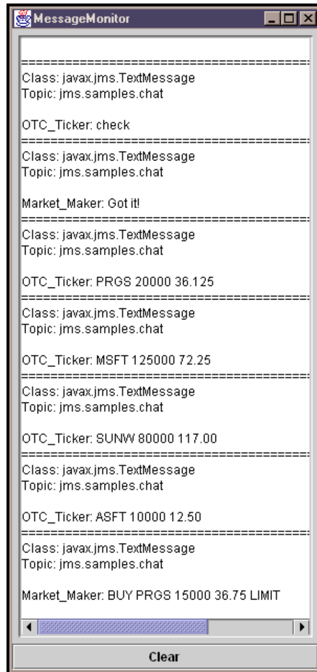


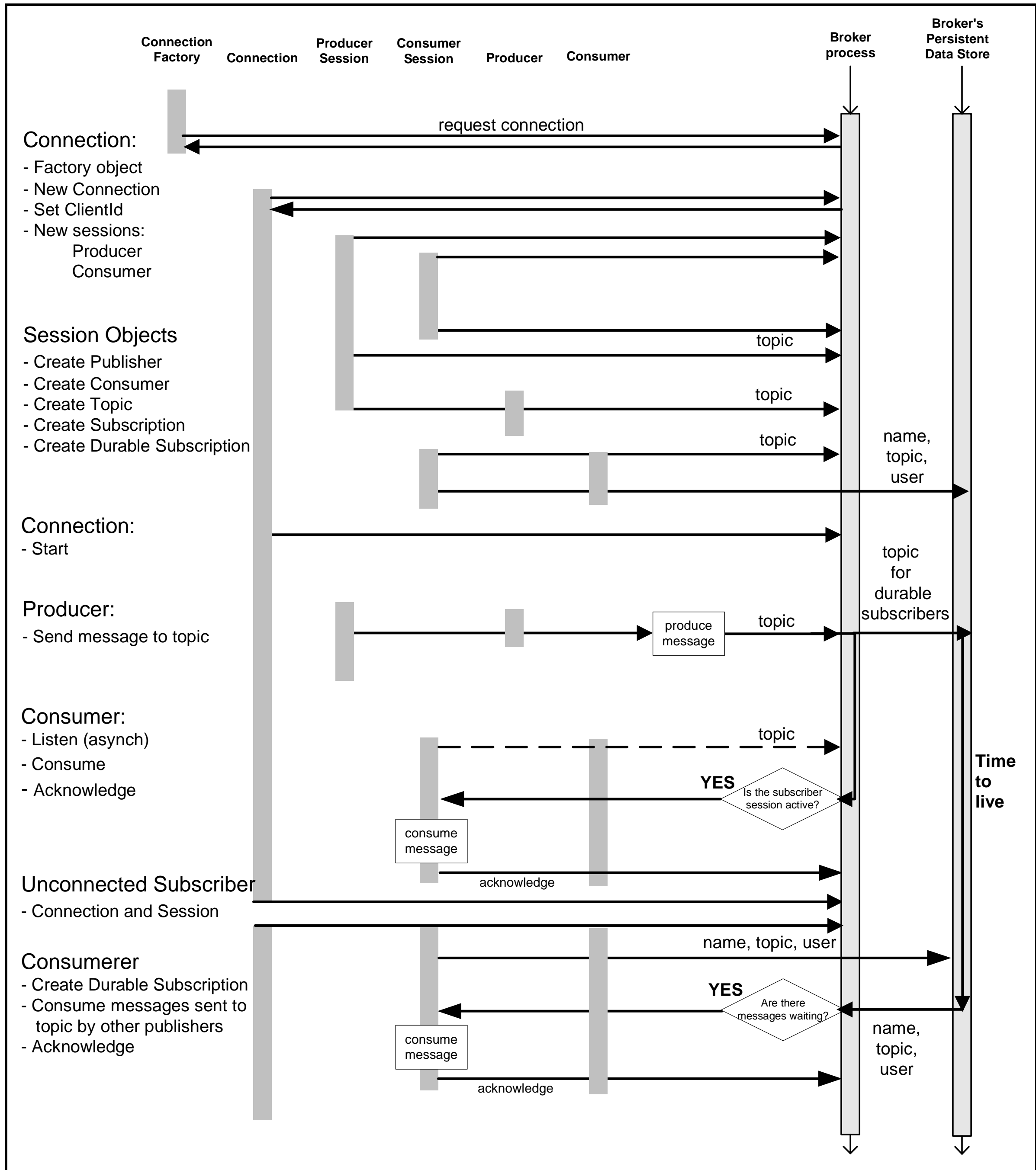
Figure 13. Message Monitor Window

Leave the `Chat` sessions and the `MessageMonitor` running and proceed to the next sample, where you will publish and subscribe to a different topic.

Durable Chat Application

When messages are published, they are delivered to all active subscribers. Some subscribers register an interest in receiving messages sent while they were inactive. These are **durable subscriptions**, permanent records in the broker's persistent storage mechanism. The `DurableChat` sample publishes its messages to the topic `jms.samples.durablechat`.

Whenever a subscriber reconnects to the topic under the name it registered for its durable subscription, all undelivered messages to that topic that have not expired are delivered in order. The administrator can terminate durable subscriptions or a client can use the `unsubscribe` method to close the durable subscription. The following figure shows what occurs when the subscriber requests extra effort to ensure delivery.


Connection Factory
Connection
Producer Session
Consumer Session
Producer
Consumer
Broker process
Broker's Persistent Data Store
request connection
Connection:
- Factory object
- New Connection
- Set ClientId
- New sessions:
Producer
Consumer
Session Objects
- Create Publisher
- Create Consumer
- Create Topic
- Create Subscription
- Create Durable Subscription
topic
topic
topic
name, topic, user
Connection:
- Start
topic for durable subscribers
Producer:
- Send message to topic
produce message
topic
Consumer:
- Listen (asynch)
- Consume
- Acknowledge
topic
YES
Is the subscriber session active?
Time to live
consume message
acknowledge
Unconnected Subscriber
- Connection and Session
name, topic, user
Consumerer
- Create Durable Subscription
- Consume messages sent to topic by other publishers
- Acknowledge
YES
Are there messages waiting?
name, topic, user
consume message
acknowledge

◆ To start DurableChat sessions:

1. Open a console window to the `DurableChat` folder:
   MQ7.5_install_root`\samples\TopicPubSub\Chat`
2. Enter:
   `..\..\SonicMQ DurableChat -u AlwaysUp`
3. Open another console window to the `DurableChat` folder, then enter:
   `..\..\SonicMQ DurableChat -u SometimesDown`
4. In the **AlwaysUp** window, type text and then press **Enter**.
   The text is displayed on both subscribers' consoles.
5. In the **SometimesDown** window, type text and then press **Enter**.
   The text is displayed on both subscribers' consoles.
6. Stop the **SometimesDown** session by pressing **Ctrl+C**.
7. In the **AlwaysUp** window, send one or more messages.
   The text is displayed on that subscriber's console.
8. In the **SometimesDown** 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 unless the messages have expired. The publisher of the message set the **time-to-live** parameter at 1,800,000 milliseconds (thirty minutes).

Note that the MessageMonitor, as a subscriber to all topics, displays all messages from both the `Chat` sessions and the `DurableChat` sessions plus system administrator messages. But, because `Chat` subscribes to **`jms.samples.chat`** and `DurableChat` subscribes to **`jms.samples.durablechat`**, `Chat` messages are only displayed in `Chat` sessions and `DurableChat` messages are only displayed in `DurableChat` sessions.

You can stop the `Chat` sessions, the `MessageMonitor`, and the `DurableChat` sessions before proceeding to the next sample.

◆ To stop sessions:

❖ In a console window, press **Ctrl+C**. The application stops.

Shared Subscriptions

A subscription can be load balanced so several subscribers can take turns receiving the latest message published—in other words, share one subscription among a group. This sample subscribes to a topic that contains a group name set off in double brackets. The name of the subscription is `[[SharedSubscriptions]]jms.samples.chat`. The group is `SharedSubscriptions` and the topic is `jms.samples.chat`. The sample sets up three subscribers and then a standard publisher.

◆ To set up three sharing subscribers:

1. Open three console windows to the `SharedSubscriptions` folder:
   MQ7.5_install_root`\samples\TopicPubSub\SharedSubsciptions`
2. In one, enter `..\..\SonicMQ SharedSubscriptions -u Member1`
3. In the next, enter `..\..\SonicMQ SharedSubscriptions -u Member2`
4. In the third, enter `..\..\SonicMQ SharedSubscriptions -u Member3`

◆ To set up a publisher:

❖ In a console window in the `Chat` folder, enter `..\..\SonicMQ Chat -u HeavyLoad`

◆ To see the effect of shared subscriptions:

❖ In the **`Chat`** window, enter `1` then press **Enter**, then `2`, and so on, up to `9`.

The three `SharedSubscriptions` windows look similar to those shown in Figure 14.

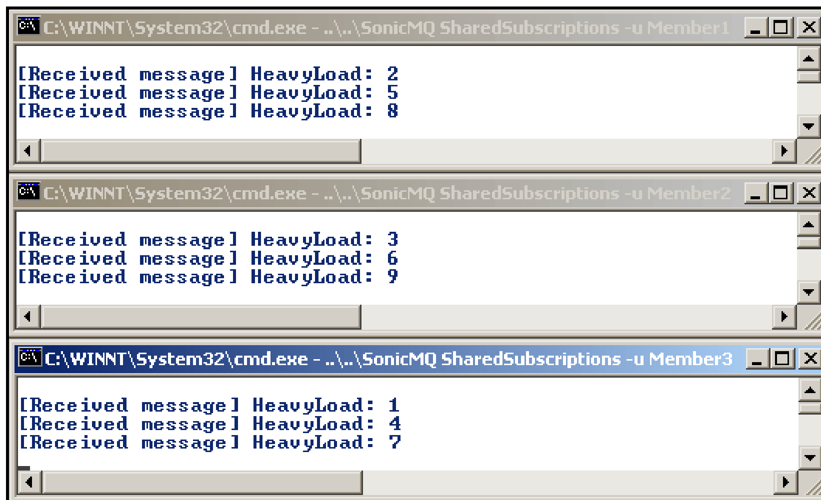


Figure 14. Three Subscribers Sharing One Subscription

The group of subscribers received all nine messages. But this is still Publish and Subscribe behavior. Other shared subscriber groups similarly distribute the subscription. All regular and durable subscribers each receive all nine messages.

You can stop all the sessions before proceeding to the next sample.

Selector Chat Application

The `SelectorChat` application demonstrates how messages can be sent to a single topic, yet subscribers can select messages that they want to see. The publisher assigns a value to a property in the message header. The subscriber examines that property by declaring the selector value (`-s`) that it wants to apply.

◆ To start SelectorChat:

1. Open a console window to the `SelectorChat` folder then enter:
   `..\..\SonicMQ SelectorChat -u Factory -s Specs`
2. Open another console window to the `SelectorChat` folder then enter:
   `..\..\SonicMQ SelectorChat -u Support -s Recalls`
3. To observe the difference selection makes, open another console window to the `Chat` folder then enter:
   `..\..\SonicMQ Chat -u NotSelective`

◆ To run SelectorChat:

1. In one of the **`SelectorChat`** windows, type text and then press **Enter**. The text is displayed in that **`SelectorChat`** window and the **`Chat`** window even though all chatters are subscribed to `jms.samples.chat`. The selector chatters are selecting messages based on different selector strings.

2. In the **`Chat`** window enter text and then press **Enter**. The message echoes in the Chat window but does not appear in either **`SelectorChat`** window because **`Chat`** is not setting a custom, user-defined property named `Department` on its messages:
   `msg.setStringProperty("Department", selector_value);`
3. Stop the **`Support`** session by pressing **Ctrl+C**.

4. Restart the session, changing the selector string, as follows:
   `..\..\SonicMQ SelectorChat -u Support -s Specs` `and press` **Enter**.
5. In either **`SelectorChat`** window, type text and then press **Enter**. The text is displayed in both **`SelectorChat`** windows. The publisher set the property value to the same value that the subscriber used to select messages.

You can stop the `SelectorChat` and `Chat` sessions before proceeding to the next sample.

Hierarchical Chat Application

SonicMQ lets an application have the power of message selectors plus a more streamlined way to often get the same result: a hierarchical topic structure that allows wildcard subscriptions. In this sample, each application instance creates two sessions, one for the specific publish topic (`-t`) and one for the wildcard subscribe topic (`-s`).

◆ To start HierarchicalChat sessions:

1. Open a console window to the `HierarchicalChat` folder.
2. Enter:
   `..\..\SonicMQ HierarchicalChat -u HQ -t sales.corp -s sales.*`
3. Open another console window to the `HierarchicalChat` folder.
4. Enter:
   `..\..\SonicMQ HierarchicalChat -u America -t sales.usa -s sales.usa`

◆ To run HierarchicalChat:

1. In the `HQ` window, type text and then press **Enter**. The text is displayed in only the `HQ` window because it subscribes to all topics in the `sales` hierarchy.
2. In the `America` window, type text and then press **Enter**. The text is displayed in both windows.

◆ To extend the sample:

1. Stop the `America` window then change it to:
   `..\..\SonicMQ HierarchicalChat -u America -t sales.usa.1 -s sales.usa.1`

   When you enter text, it displays in this window but not in the `HQ` window because that subscription is limited to only the second level of the topic hierarchy.
2. Stop the `HQ` window then change it to:
   `..\..\SonicMQ HierarchicalChat -u HQ -t sales.corp -s sales.#`
3. Enter text in the `America` window.

   The text displays in both windows. The `HQ` session is subscribed to all topics and all levels under `sales`.

You can stop the `HierarchicalChat` sessions before proceeding with the next sample.

XML Messages

XML documents are composed of tagged text blocks that provide a logical way to interpret the content of the message.

XML Chat Under the Document Object Model

An XML processor, acting on behalf of an application, interprets the data structure by parsing XML and Document Type Definitions (DTDs)—sets of rules that define the elements used in a document and their relationships.

The Document Object Model (DOM) is an application programming interface (API) for valid HTML and well-formed XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

With the Document Object Model (DOM), programmers can build documents, navigate their structure, and add, modify, or delete elements and content. An important objective for the Document Object Model is to provide a standard programming interface that can be used in a wide variety of environments and applications.

◆ **To start an XMLDOMChat Publisher:**

❖ Open a console window to the `XMLDOMChat` folder, then enter:

```
..\..\SonicMQ XMLDOMChat -u Catalog_Update
```

◆ **To start an XMLDOMChat Subscriber:**

❖ Open another console window to the `XMLDOMChat` folder, then enter:

```
..\..\SonicMQ XMLDOMChat -u Aggregator
```

◆ **To start a Chat Subscriber:**

❖ Open a console window to the `Chat` folder, then enter:

```
..\..\SonicMQ Chat -u Just_Text
```

◆ To send an XML Message:

❖ In the Catalog_Update window, type text—for example, **Gadget 1.00**—and then press **Enter**. The text entry is formulated into a simple, yet complete XML document:

- The **Catalog_Update** and **Aggregator** windows display the input as Document Object Model Element nodes:

```
[XML from 'Catalog_Update'] Gadget 1.00
ELEMENT: message
 |--NEWLINE
 +--ELEMENT: sender
     |--TEXT_NODE: Catalog_Update
 |--NEWLINE
 +--ELEMENT: content
     |--TEXT_NODE: Gadget 1.00
```

- The **Just_Text** window, while it subscribes to same topic, does not invoke the XML parser. It simply displays the XML data:

```
<?xml version="1.0"?>
<message>
  <sender>Catalog_Update</sender>
  <content>Gadget 1.00</content>
</message>
```

◆ To send a TextMessage to the XML sessions:

❖ In the Just_Text window, type text—for example, **Hello**—and then press **Enter**. The TextMessage is sent to the subscribers:

- Both XMLChat sessions, **Catalog_Update** and **Aggregator**, handle the instance of a **TextMessage** as a simple text message because the XML parser was not invoked:

  ```
  [TextMessage] Just_Text: Hello
  ```

- The Chat session for **Just_Text** displays the text in its usual way:

  ```
  Just_Text: Hello
  ```

XML Chat Under SAX

Simple API for XML (SAX) provides an efficient mechanism for parsing XML, and is useful for parsing large XML documents. Servlets and network-oriented programs can use this to parse XML documents because of its speed. SAX is also one of the least memory-intensive mechanism currently available for handling with XML documents.

In this example, you run an application to publish and subscribe to a specified topic. Other sample applications might also publish to this topic and view the messages published to this topic. However, those sessions view messages as `javax.jms.TextMessage`, a superclass of the `progress.message.jclient.XMLMessage` that this application uses. As a result, those sessions see the message as it was published.

◆ To start several XMLSAXChat sessions:

1. Open a console window to the `XMLSAXChat` folder, then enter:
   `..\..\SonicMQ XMLSAXChat -u Wholesaler`
2. Open another console window to the `XMLSAXChat` folder, then enter:
   `..\..\SonicMQ XMLSAXChat -u Retailer`

◆ To send an XML Message:

❖ In the **`Wholesaler`** window, type text—for example, **`Hello`**—and then press **Enter**. The text entry is formulated into a simple, yet complete XML document.

The **`Wholesaler`** and **`Retailer`** windows display the input as raw XML data:

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

All the samples that are running can be stopped now.

Queue Point-to-point Samples

This set of samples explore the Point-to-point messaging model. These samples progress:

- From demonstrating basic PTP messaging behavior,
- To showing a GUI window that views available messages in a queue,
- To showing how a subscription can be active when the application is not connected,
- To showing how applications can attempt to reconnect when their connections drop,
- To showing how multiple part messages are handled,
- To showing how transacted sessions stage messages sent and received into batches,
- To showing how messages can request that the receiver reply to a temporary location.

About Queues

In SonicMQ, a queue cannot be created dynamically from a client session. The administrator must create a static queue before a queue can be used by a client. The following samples require that the sample queues were set up in the broker persistent storage mechanism when SonicMQ was installed. These queues are set up in a default installation and should be available. See the *Progress SonicMQ Configuration and Management Guide* for information about queues.

Many of the queue-based samples require that you enter the queue receiver (**-qr** *queuename*) and the queue sender (**-qr** *queuename*) as parameters. Unlike the publishers and subscribers in the previous set of samples, using one queue likely means that each sender receives its own message. So when the instructions request you to run the same application with queue parameters, notice that the queue where one sends is the queue where the other one receives.

Talk Application

In `Talk`, you start two console sessions, each with a consumer and a sender where one's sender queue is the other's consumer queue. Then, when you type text and press **Enter**, the message is sent only to the indicated `Talk` partner. If you start several consumers, only one of them receives a message that is produced and sent to the queue.

The **`-qr`** parameter indicates the queue where the application will receive messages. The **`-qs`** parameter indicates the queue where the application will receive messages.

◆ To start Talk:

1. Open two console windows to the `QueuePTP\Talk` folder.
2. In one window, enter:
   ```
   ..\..\SonicMQ Talk -u Sales -qr SampleQ1 -qs SampleQ2
   ```
3. In the other window, enter:
   ```
   ..\..\SonicMQ Talk -u Orders -qr SampleQ2 -qs SampleQ1
   ```

◆ To run Talk:

1. In the **`Sales`** window, type **`Here is an order.`** and then press **Enter**.
   The **`Orders`** window displays the text.
2. In the **`Orders`** window, type **`Order is confirmed.`** and then press **Enter**.
   The **`Sales`** window displays the text.

Leave the `Talk` sessions running and proceed to the next sample where you will start a Java window that will browse the queue.

Queue Monitor

The QueueMonitor displays messages in a queue. This is different from the Topic Pub/Sub sample, MessageMonitor. The two monitors underscore fundamental differences between the two messaging models. For example:

- **What messages are displayed?** The QueueMonitor displays messages that are not yet delivered. The MessageMonitor displays published messages it received.
- **When does the display update?** The QueueMonitor updates when you click the **Browse Queues** button. The MessageMonitor updates whenever a new message is received.
- **When does a message no longer display?** The QueueMonitor no longer displays messages when they are delivered (or expire.) The MessageMonitor displays messages that have been delivered; when the display window is cleared the messages are gone.
- **What happens when the broker and monitor are restarted?** Queued messages are in the broker persistent storage mechanism if they were sent with a delivery mode of PERSISTENT. These messages are redisplayed when the broker and the QueueMonitor restart. The MessageMonitor starts receiving new messages when the application starts; messages that were displayed before it restarted are gone.

◆ To start QueueMonitor:

1. Open a console window to the QueuePTP\QueueMonitor folder.
2. Enter: **..\..\SonicMQ QueueMonitor**

   The console displays the list of queues that it will monitor and then opens the **QueueMonitor** window.

◆ To stop one Talk Session so that there are messages on a queue:

This sample application uses a properties file to specify one or more queues that will be monitored. As the sample application is set to browse only **SampleQ1**, stop the **Sales** window so that messages sent to **SampleQ1** stay in the queue.

❖ In the **Sales** window, press **Ctrl+C**. The application stops.

◆ To queue messages and browse the queue:

1. In the **Order** window, type **First** and then press **Enter**.
2. Type **Second** and then press **Enter**.
3. Type **Third** and then press **Enter**.
4. In the **QueueMonitor** window, click **Browse Queues** to scan the queue.

5. The QueueMonitor lists the messages in SampleQ1, as shown in Figure 15.

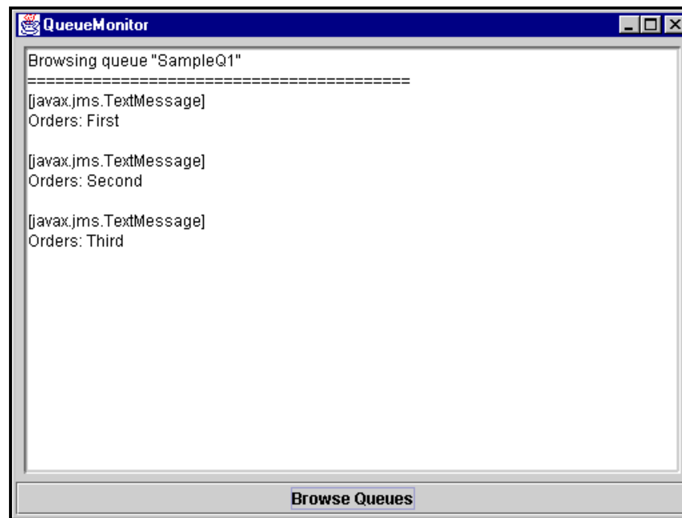


Figure 15. QueueMonitor Window

◆ To receive the queued messages:

The messages waiting on the queue are delivered to the next consumer to receive from that queue.

Warning If you do not perform this procedure, the stored messages are received in the next sample application that receives on that queue.

❖ In the Sales session window or a new console window in the Talk folder, enter:

```
..\..\SonicMQ Talk -u Sales -qr SampleQ1 -qs SampleQ2
```

The queued messages are all delivered to the sole consumer on the queue.

You can close the QueueMonitor window to conserve resources. You can also stop the Talk sessions.

Reliable Talk Application

The ReliableTalk sample ensures the robustness of the connection by monitoring the connection for exceptions, and re-establishing the connection if it is dropped.

◆ To start a ReliableTalk session:

1. Open a console window to the QueuePTP\ReliableTalk folder, then start a session that sends and receives on the same queue by entering:

   `..\..\SonicMQ ReliableTalk -u EverReady -qr SampleQ1 -qs SampleQ1`

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

   The text is displayed, preceded by the user name. The message was sent from the client application to the message broker and then returned to the client as a consumer 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.

4. Restart the broker by using its Windows **Start** menu command or run the **bin\startcontainer** script.

   The **ReliableTalk** application reconnects.

You can stop the ReliableTalk session before proceeding to the next sample.

Working with MultiPart Messages

SonicMQ extends its message type to allow creation and handling of messages that are an assemblage of parts. Multipart messages are common in mail applications—pictures, documents, text, and executables are all packaged as attachments to a mail message. This packaging is common in business-to-business message packaging such as SOAP 1.1 with attachments.

◆ To run the Multipart message sample:

❖ In a console window in the QueuePTP\MultipartMessage folder, type
`..\..\SonicMQ Multipart -u aUser`
and press **Enter**.

The sample generates a four part message as follows:

```
sending part1..a TextMessage
sending part2..some bytes
sending part3..a simple text string
sending part4..a Readme file
```

The consumer in the sample takes the message from the queue and discovers that it is a multipart message. The consumer processes each part of the message sequentially:

```
    ******* 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 3
Part.contentType = text/plain
Part.contentId = CONTENTID3
...size :  37
...content :
a simple text string to put in part 3
    --------end of part 3
    --------Beginning of part 4

Part.contentType = text/plain
Part.contentId = CONTENTID4
...size :  884
...content :
Multipart Sample
====================
This sample application is to show that you can send and
receive multiple parts in a batch using MultipartMessage.
...
- This sample program uses classes in activation.jar of
JavaBeans Activation Framework (from Sun).
Make sure that activation.jar is in CLASSPATH
    --------end of part 4
    ******* End of MultipartMessage ******
```

The four parts to this message are each handled according to their MIME content-type. MIME types that are viable JMS message types are mapped to that JMS message type. When the content type is `text/xml`, the message is mapped to SonicMQ's `TextMessage` extension, an `XMLMessage`.

All the samples that are running can be stopped now.

Transacted Talk Sessions

Transacted messages involve a session where groups of messages are stored in the queue on the broker until the transactions are either committed as a set or rolled back as a set. The TransactedTalk sample commits or rolls back messages that are sent but does not affect messages that are received. The application creates two sessions—sendSession and receiveSession—on one queue connection.

When the notice to **commit** is sent, the series of messages held by the broker is released sequentially to consumers, though not as a bound set.

If, at any point, the notice to **roll back** is sent, the broker is instructed to clear the series of messages from the queue.

After either a commit or a rollback, the session starts a new transaction.

Note When you start these sessions after running the **SelectorChat** samples, you might find that the queue consumers receive messages that are artifacts of that sample.

◆ To start TransactedTalk sessions:

1. Open a console window to the QueuePTP\TransactedTalk folder, then enter:
   ```
   ..\..\SonicMQ TransactedTalk -u Accounting -qr SampleQ1 -qs SampleQ2
   ```
2. Open another console window to the TransactedTalk folder, then enter:
   ```
   ..\..\SonicMQ TransactedTalk -u Operations -qr SampleQ2 -qs SampleQ1
   ```

◆ To build a transaction and then commit it:

1. In one of the TransactedTalk windows, type text and then press **Enter**.

   Notice that the text is not displayed in the other TransactedTalk window.
2. Again type text in that window and then press **Enter**.

   The text is still not displayed in the other TransactedTalk window.
3. Enter **OVER**

   All of the lines you entered are sent to the designated queue and then delivered to consumers with the transaction setting marked **TRUE**. The transaction buffer is cleared. Subsequent entries will accrue into a new transaction.

◆ **To build a transaction and then roll it back:**

1. In one of the `TransactedTalk` windows, type text and then press **Enter**.
2. Type additional text in that window and then press **Enter**.
3. Enter **`OOPS!`**

   Nothing is published. The transaction buffer is cleared.
4. Enter **`OVER`**

   No messages are sent.

   Subsequent entries will accrue into a new transaction. After a commit or rollback, the session begins accruing messages again for the next transaction.

You can stop these sessions before proceeding to the next sample.

Request and Reply

In the Point-to-point messaging models, the producer of a message can support long-lived messages that persist in a queue's datastore. The queue can be browsed to see if the message is still on the queue, but once the message is delivered, there is no implicit mechanism for reporting to the producer that the application or client received it.

Much like its Publish and Subscribe counterpart, requesting a reply when a message is sent sets up a **temporary queue** for that request; then the header information specifies that the consumer send a reply to the sender of the original message. A correlation identifier can be used to coordinate the activities.

In this simple example, a replier is set up to simply receive text and send back the same text as either all uppercase characters or all lowercase characters to the temporary queue set up for the reply.

◆ **To set up PTP Request Reply sessions:**

❖ Open two console windows to the `QueuePTP\RequestReply` folder.

◆ **To start the PTP replier:**

❖ Enter **`..\..\SonicMQ Replier -u QReplier`** in one of the windows.

◆ **To start the PTP requestor:**

❖ Enter **`..\..\SonicMQ Requestor -u QRequestor`** in the other window.

◆ **To test a PTP request and reply:**

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

The `Replier` window reflects the activity, displaying:

[Request] QRequestor: AaBbCc

The replier performs its conversion (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

This kind of message processing adds considerable performance overhead to messaging systems but is appropriate to financial transactions that must be audited and reconciled, resulting in overall savings in the time and cost of audits.

Stopping Client Sessions and the Broker

You have completed all the samples in this book.

◆ **To stop client sessions and the broker:**

1. In each active client console window, press **Ctrl+C**, and then close the window.
2. In the broker process window, press **Ctrl+C**. Wait for the console to complete its shutdown, and close its window.

Other Samples Available

Now that you have explored the basic samples, you might be interested in some of the other samples that are part of the SonicMQ package. Many of these samples require special setup steps to explore them. These samples are described in the following SonicMQ documents.

SonicMQ Application Programming Guide

The *Progress SonicMQ Application Programming Guide* goes through the samples included in this guide but with a different purpose. The Point-to-point and Pub/Sub samples that explore a feature and exercises are offered that extend the sample scope and the sample applications.

Other samples in the *Progress SonicMQ Application Programming Guide* include:

- **Client Plus** — The features that are available when you perform a SonicMQ Client Plus installation are explored: local client persistence and handling large messages through recoverable file channels.
- **Distributed Transactions** — The XA resources in SonicMQ provide the functionality to explore global transactions in a standalone sample.
- **JNDI Lookup** — `JNDITalk` demonstrates the lookup of administered objects in the local JNDI store.
- **JNDI SPI** — The Sonic service provider implementation (SPI) for the Java Naming and Directory Interface (JNDI).

Example for Fault Tolerant Client Connections

An example lets you modify any application to add code that will enable fault tolerant connections and display the connection URLs of the brokers. When the `Chat` sample is modified this way, it lets you chat with a single broker that enables continuously available connections.

SonicMQ Deployment Guide

The *Progress SonicMQ Deployment Guide* pulls together the programming and administrative functions that enable sophisticated message routing and security. These samples include:

- **Dynamic Routing queues** — The GlobalTalk (PTP) sample demonstrates dynamic routing queues. See the chapter "Multiple Nodes and Dynamic Routing."
- **Advanced security** — Presents an overview of the concepts in the advanced security samples for login and external authentication.
- **HTTP(S) Direct** — A variety of HTTP Direct samples plus HTTPS Direct samples showing inbound and outbound HTTP Direct messaging, and basic authentication. The HTTP Direct for JMS and HTTP for SOAP variations also have samples.

Example of a Secure Socket Layer (SSL) implementation

See the "SSL and HTTPS Tunneling Protocols" chapter for an example of how you can set up a broker to accept SSL connections, and how to modify the startup script of the Talk sample so that the sample application uses SSL for its connection on the defined acceptor.

Example of Security Enabled Dynamic Routing

See "Multiple Nodes and Dynamic Routing" in that book for an example of how you can set up multiple brokers and security to realize secure dynamic routing across nodes.

Example of Management Communications Through Dynamic Routing

See the "Multiple Nodes and Dynamic Routing" chapter in that book for an advanced example of how a broker can provide routing of management communications for multiple containers through a routing connection to the domain's management node.

Example of Replicated (High Availability) Brokers

Continuous availability of messaging brokers is explored by setting up brokers as a primary/backup pair. When they are running and replicating, the active broker is stopped and the standby broker fails over. Then run the fault tolerant example described in the *Progress SonicMQ Application Programming Guide*, to see the client application seamlessly continue its session on the broker that becomes active.

Examples of Fault Tolerant Management Services

Management components can be fault tolerant. The example in the *Progress SonicMQ Deployment Guide* transforms a basic management framework installation into completely independent primary and backup management locations that provide replication of management activities and failover from one site to its standby.

SonicMQ Administrative Programming Guide

The *Progress SonicMQ Administrative Programming Guide* has the following management sample applications:

- **Configuration API** — Describes samples of creating configuration objects in Java and JavaScript.
- **Runtime API** — Describes the JMX Dynamic MBean sample, and the Java and JavaScript samples of runtime actions such as shutting down and clearing queues.
- **Advanced Security SPIs** — Describes the sample applications for advanced security Login, Authentication, and Management.

Chapter 4 Next Steps

When you have finished reading this book:

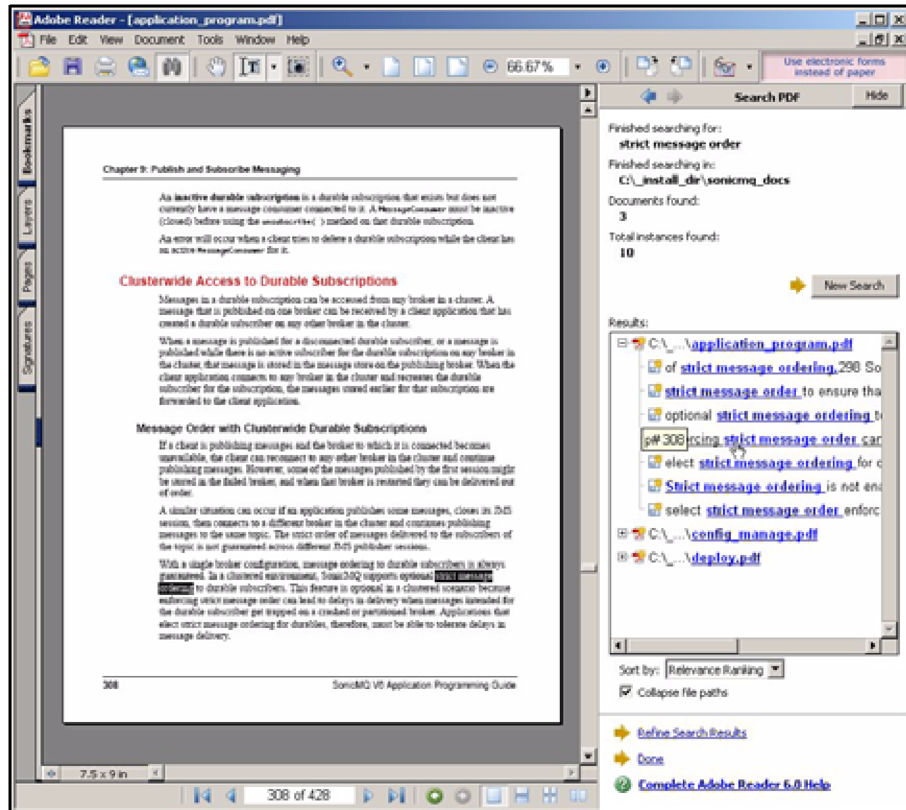
- **Look at the other SonicMQ Guides and their samples** — Explore the other books in the SonicMQ documentation set to learn more about the concepts and features described in this book. Here are two suggested tracks for exploration:

    1. **Administrator's track** — Start with the *Progress SonicMQ Deployment Guide* for in depth information on the concepts of component distribution, clusters, load balancing, the Dynamic Routing Architecture®, the Continuous Availability Architecture™, and security. Then use the *Progress SonicMQ Configuration and Management Guide* for information on configuring, monitoring, and managing containers and other components. The *Progress SonicMQ Administrative Programming Guide* shows how to create management applications. Use the *Progress SonicMQ Installation and Upgrade Guide* to learn about component distributions through the installer in concert with the Sonic Management Console. Tune your deployment with tips and techniques described in the *Progress SonicMQ Performance Tuning Guide*.

2. **Application developer's track** — Start with the *Progress SonicMQ Application Programming Guide* to expand the scope of the sample applications to provide you with experience in the product features as well as a starting point for creating your own solutions. Learn standard JMS plus the SonicMQ extensions that build messaging applications. Explore the examples that extend the sample applications to learn more about design patterns you can apply such as fault tolerant client connections. See Part IV of the *Progress SonicMQ Deployment Guide* to learn about HTTP Direct to integrate HTTP Web server functionality and HTTP outbound into your suite of applications. Look for tuning information for clients in the *Progress SonicMQ Performance Tuning Guide*.

- **Select a project to evaluate** — Choose a business integration problem to define and implement. Make it significant enough to measure its success but keep it constrained enough to let you explore alternative approaches to see what's best for your business.
- **Contact Sonic Software** — The Sonic Software Technical Support group, the Sonic Software Professional Services organization, and SonicSynergy partners are ready to provide help in defining and implementing your deployment. Refer to "Worldwide Technical Support" on page 13 for information about their Web site address and contact email addresses and telephone numbers.
- **Visit the Sonic Software Web site** — Browse to `sonicsoftware.com` for current company and product information, events, training sessions, and more.
- **Search the Documentation** — Adobe Reader 6.0 is recommended to search for key words or phrases. You can perform a search across multiple books, see the number of hits (each with a context hint), and then go to any link.

The following screen capture is an example of an Adobe Reader search in a SonicMQ documentation folder:



Glossary

Activation Daemon A configuration object that provides a way to launch and monitor new child containers as spawned processes of the container hosting the Activation Daemon. This allows new containers to be launched by remote administrative clients without the administrator having to log on to that host.

Administrator The one user created in a new SonicMQ Authentication Domain. The Administrator user has unique capabilities under management permissions as the only user that can control whether enforcement is enabled and the only user that always has all permissions.

Administrators The group created by an installation of a new SonicMQ Authentication Domain that enables its users to make management connections and perform administrative and operational tasks. The Administrators group has a default set of permissions when management permissions is enabled.

Agent Manager A configuration object and framework component that maintains a record of the runtime status of the containers and components in its domain.

Annotations A property of every configuration object that allows for user metadata text to be stored with the object's configuration.

Application Permission An administrator can define patterns that grant or deny permissions to users or groups for sending and receiving from named queues, publishing and subscribing to named topics, and routing to named remote nodes.

Auditing In Sonic, the auditing feature records configuration changes, operational actions, and, when using management permissions, the user that made the change or performed the action.

Authentication Domain A configuration object that stores users and groups used for authenticating users attempting connection to brokers that bind to the specified authentication domain.

Authorization Policy A configuration object that maintains the Access Control Lists that define the patterns evaluated to determine an application's access to destinations, and Quality of Protection rules that specify the application's requirements for encryption and integrity on messages sent to destinations that match the defined patterns.

Backup Broker A configuration object derived from a broker. The peer brokers, the primary and its backup, have a replication connection between them to replicate client states and data so that, in the event of the active peer failing, the one standing by can promptly take on the active role.

Broker A configuration object that provides services to its clients. Every SonicMQ broker performs the following functions:

- Enables acceptors for simultaneous multi-protocol support for connections (TCP, HTTP(S) Tunneling, HTTP(S) Direct, and SSL)
- Enforces authentication and authorization of users and routings
- Receives and delivers messages
- Manages its persistent data store

C client Any of the non-Java SonicMQ clients that provide a JMS client interface. The C clients include pure C++, ANSI C, COM, and C#. Applications created in these clients interface with a SonicMQ broker with many of the same behaviors as a Java client.

CAA Progress Sonic's Continuous Availability Architecture.

CAA-FastForward The Progress Sonic non-persistent delivery mode for messages in fault-tolerant deployments. Messages sent with this qualified nonpersistent delivery mode, `NON_PERSISTENT_REPLICATED`, are protected against message loss due to broker failures by replicating the messages to the standby broker.

Certificate Manager The Progress Sonic SMC tool that manages certificates used in SSL-based communications.

Certificates store A configuration object (also known as a keystore) that stores the certificates, keys, and related data in the file system in a serialized, encrypted format.

Client Persistence Client-based logging of messages sent when a broker connection is not active. This enhances delivery guarantees and provides disconnected operation.

Cluster A configuration object whose members are brokers. The brokers all use the same routing node name, routing definitions, and authenticate users in the same authentication domain. Each broker in the cluster communicates directly with every other broker in the cluster to expedite clusterwide access to topics and queues.

Clusterwide Access The ability of client applications to reconnect to any broker in a cluster to either activate a durable subscription and take delivery of the stored messages on any broker in the cluster, or access a clustered queue from a different broker from the one where the sender enqueued the message.

Collection A logical grouping of either components or containers that is maintained as a configuration in the Directory Service.

Collections Monitor A configuration object that performs monitoring functions across the components of a component collection. Collections Monitors can perform notification forwarding, notification monitoring, and metrics aggregation.

Component A configuration object in a configuration domain that is hosted in a management container at runtime.

Component Collection A configuration object that groups components for organizational and monitoring purposes. Operations performed on a collection are performed on all of the components in the collection.

Configuration Domain A domain manager. The frameworks components and management brokers that enable maintenance and management of deployed configurations.

Connection A communications channel between a client and a broker.

Consumer Recipient of messages sent by a producer to a broker destination. The consumer binds to a broker destination to receive a message. See also Destination and Producer.

Container See *management container* and *ESB container.*

Container Boot File A file used to start a container. It specifies the configuration identity of the container and its cache from which to retrieve the full container configuration information.

Container Collection A configuration object that groups selected containers for a single view of composite runtime state, and for initiation of restart, shutdown, and log clearing operations on all its containers.

Continuous Availability Architecture The Progress Sonic set of technologies that implement fault tolerance for clients, containers, brokers, and framework components. This fault tolerance design provides such high availability—by using a deployment architecture that has no single point of failure—that it is realized as continuous availability. That means that whenever an active broker component, framework component, or service container experiences a failure, a standby container or component is ready and waiting to take over for the lost one and reconnect to fault-tolerant clients seamlessly.

Control Number The text string that a customer or evaluator must enter at installation time in order to install the product. Sometimes referred to as the license key or control key.

Dead Message Queue A system-defined queue that stores messages configured to be preserved after they expire, become indoubt, or unroutable.

Delivery Mode The message producer parameter that specifies to the broker whether the message is nonpersistent or persistent. A message's delivery mode is effective throughout its lifespan.

Destination An object that encapsulates addresses and configuration information on a broker. Message producers send messages to destinations and message consumers receive messages from destinations. A destination is either a topic or a queue.

Directory Service A configuration object and framework component that manages a directory store of persistent configuration information for a domain as a continuously available service through which configuration information is accessible by management tools and runtime containers.

Domain In Progress Sonic, a domain is the term used for an 'authentication domain' and a 'configuration domain. When the term is not qualified, it typically refers to a configuration domain.

Domain Manager The set of components that establishes a unique configuration domain. A domain manager includes an Agent Manager, a management container, a management broker, and a Directory Service.

Durable Subscription A subscription where the client registers an interest in receiving all messages published on a topic, even when the client connection is not active. The broker manages the durable subscription, retaining all messages from the topic's publishers until the message is acknowledged by the durable subscriber, or until the message expires.

Dynamic Routing Architecture The patented Progress Sonic technology that dynamically routes messages between software application programs using named routing nodes and named messaging queues.

Failover The behavior of Progress Sonic fault-tolerant components that assume the active role when their peer is perceived to have failed:

- In brokers, the technique of replicated brokers to switch from a standby broker to an active broker in the event of a network or broker failure.
- In client connections, the ability to maintain state and resume connection to another network or peer broker that has also maintained client state and current data.
- In management framework components, the ability for the framework components and Directory Service store to switch to its set of backup components and the replicated Directory Service store to resume current configurations and runtime management seamlessly.
- In management containers, the ability of a container to start its components when its peer is no longer active.

Fault Tolerance SonicMQ fault tolerance allows containers, brokers, and management components to continue to function despite failures. Fault tolerant connections allow clients to reconnect after a broker or network recovery or failover without loss of data.

Flow Control The ability of a broker to refrain from accepting messages sent to a destination when the physical resources allocated to the destination are below a specified level.

Flow to Disk A feature that allows publishers to continue publishing even after the buffers in the broker are full.

Framework Components Configuration objects in the current version of the Sonic Management environment that are components of the domain's framework, such as the domain's Agent Manager and Directory Service.

Globally Advertised Destination A queue that becomes known to a broker when two routing nodes connect and advertise to each other the names of the global queues that each supports.

HTTP Direct A set of protocol handlers that allows applications to communicate directly with SonicMQ brokers through the exchange of HTTP payloads without requiring the use of SonicMQ client software on the client system.

Integrity The Quality of Protection setting that can confirm whether a message has changed in transit.

Interbroker Connections The connections between a broker and its fellow members of a SonicMQ cluster.

Logger A configuration object that collects monitored notifications and metrics from a Collections Monitor and logs that data to configured log destinations.

Management Container A configuration object that is a Java process that hosts one or more components in a controlled execution environment (JVM). The hosted components of a container include framework components, and messaging infrastructure components. A management container establishes management communications with its domain's management and maintains a local cache of its configurations. A container provides the services that components require to boot and obtain their configuration, expose their management API, and log messages to the container console and its log file.

Management Framework Fault Tolerance A domain manager that is maintained in two locations where one is active, accepting administrative updates and replicating its store to its peer, and the other is standing by, prepared to continue the management functions when the currently active fails.

Management Permissions Domain settings that grant or deny specified administrative principals the ability to maintain configurations and perform runtime operations.

Message Consumer A client application that has a connection to a broker and is receiving messages from a queue or topic.

Message Producer A client application that has a connection to a broker and is sending messages to a queue or publishing messages to a topic.

Message Type In SonicMQ, there are eight message types:

- Message — Basic message; no body is required.
- TextMessage — Standard java.lang.String.
- XMLMessage — SonicMQ-specific derivation of the Text type; specifically attuned to interpretation of the text as XML-tagged text.
- ObjectMessage — Serializable Java objects.
- StreamMessage — Stream of Java primitives, read sequentially.
- MapMessage — Set of name-value pairs where values are Java primitives.
- BytesMessage — Stream of uninterpreted bytes.
- MultipartMessage — SonicMQ-specific derivation of the Message type; uses the data handlers 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 can treat existing JMS messages as parts of a multipart message and include one multipart message inside another.

MultiTopic A destination that combines multiple underlying component topics into a single destination.

Node A single broker or a cluster of brokers.

Password Based Encryption Tool The Progress Sonic command-line tool that encrypts and decrypts boot files. Also referred to as the pbetool.

Permission The term permission refers to ‘application permissions’ and to ‘management permissions.’

Point-to-point The messaging model in which a producer (sender) delivers a message to a specified destination (queue) at the broker, placing new messages at the back of the queue. Consumers (receivers) can either receive the first message in the queue (as qualified by specified message selection rules) or browse through all the messages in the queue. When a consumer receives a message, that message is removed from the queue. Point-to-point (PTP) messaging is referred to as one- to-one communication because, although multiple consumers can access a queue, each message is received by one and only one consumer.

Privacy The Quality of Protection setting that encrypts the message. Privacy also includes integrity.

Producer A client that sends messages to a specified destination in the broker. These messages are received by consumers from the broker destination. The producer packages (when appropriate, also encrypts the message body), and identifies the service level and protection for the outbound message.

Publish and Subscribe The messaging model in which a producer (publisher) delivers a message to a specified destination (topic) at the broker. Consumers (subscribers) subscribe to a topic to receive messages published to that topic. Publish and subscribe (Pub/Sub) messaging is referred to as one-to-many communication because all consumers subscribed to a topic receive all messages published to that topic.

Publisher A message producer in Pub/Sub messaging.

Quality of Protection The setting in the Authorization Policy that determines the security protection to be applied to messages sent to the specified destination.

Quality of Service The range of service levels that provide commitment to secure and accurate message delivery at the expense of time and system resources. The advantage of a broad QoS is that simple information updates can be expedited and crucial business information can be guaranteed and secure.

Queue In PTP messaging, messages are produced to named queues.

Recoverable File Channel A unidirectional stream of information from a JMS client to another JMS client. This peer-to-peer type of transfer, similar to FTP, sends messages to the recipient by fragmenting the larger file into smaller parts and keeping track of the transfer progress and status.

Recovery Log The files that records events and messages that must be persisted in the event of a broker failure.

Replication Connection One or more reserved connection definitions between fault-tolerant broker peers or fault-tolerant Directory Service peers, over which data is synchronized in anticipation of the standby taking over for the active when it fails.

Sender The message producer in Point-to-point messaging.

Session A single thread of activity in JMS messaging.

Sonic Event Monitor The Progress Sonic product that, under its own license, enables Logger configuration objects.

Sonic Management Console (SMC) The Progress Sonic graphical tool that performs and visualizes configuration, management, and monitoring of functions in one or more domains.

Sonic Management Environment The set of management tools and functionality that lets you remotely configure, monitor, and manage all aspects of a distributed SonicMQ deployment.

Subscriber A message consumer in Pub/Sub messaging that receives messages when it is active and has specified an interest in a topic.

Template A configuration pattern used only for creation of derived, deployable configurations.

Template Character Substitute characters used in definition of a hierarchical name, to be applied as wildcards when evaluating a namespace.

Topic In Pub/Sub messaging, messages are produced to dynamic named topics.

WebService Protocol A configuration object that enables a SonicMQ broker to process SOAP messages as required by the WS-Security and WS-ReliableMessaging specifications. The protocol can be used outside the context of these standards.