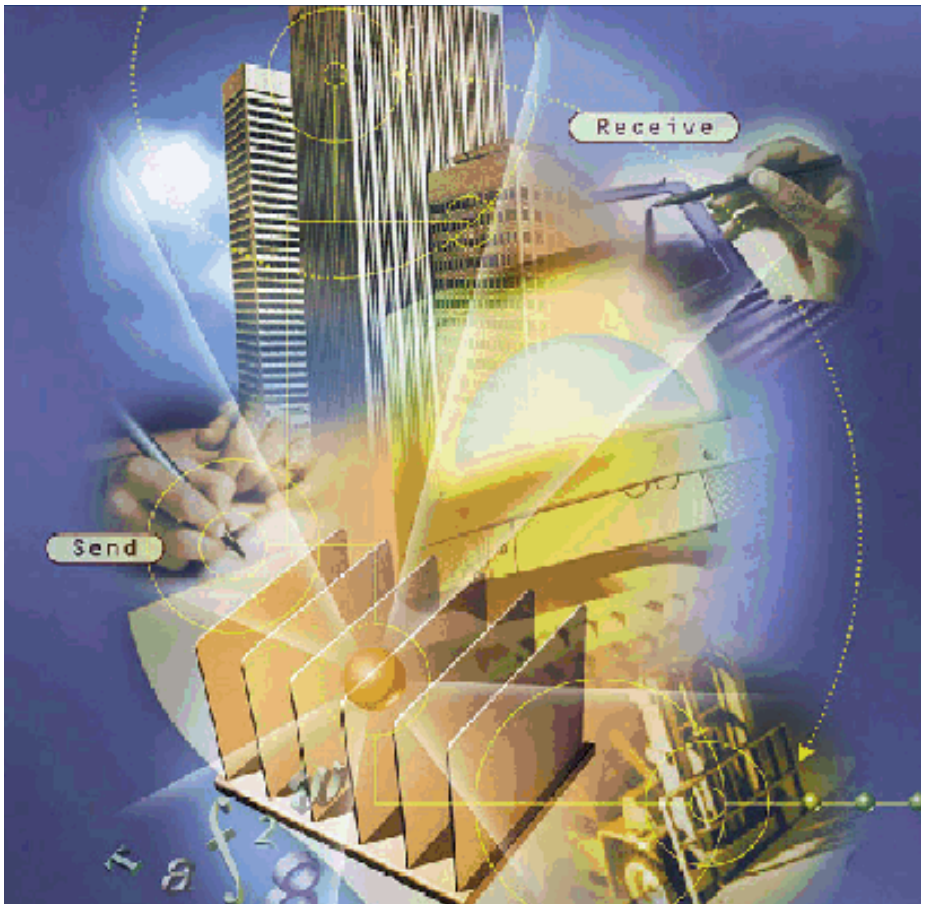
Progress®
SonicMQ™

Getting Started with SonicMQ



Printed in U.S.A.
November 2000

Contents

List of Figures

List of Tables

Preface

This Preface contains the following sections:

- "About This Manual" describes this manual and its intended audience.
- "Conventions in This Manual" describes the text formatting, syntax notation, and flags used in this manual.
- "Available Documentation" describes the printed and online documentation that accompanies SonicMQ.
- "Worldwide Technical Support" provides information on contacting technical support.

About This Manual

Progress SonicMQ is a fast, flexible, scalable E-Business Messaging Server designed to simplify the development and integration of today's highly distributed enterprise applications and Internet-based business solutions. SonicMQ is a complete implementation of the Java Message Service v1.0.2, an API for accessing enterprise messaging systems from Java programs.

This book provides an overview of the Progress SonicMQ software for developers, development managers, and enterprise IT managers.

After presenting the features and concepts of SonicMQ, this guide details the steps to install, set up, and start SonicMQ Developer Edition. Sample programs demonstrate implementations of SonicMQ and usage scenarios.

The glossary of terms used throughout SonicMQ is included in this book.

Conventions in This Manual

In this section, you will find a description of the text-formatting conventions used in this manual and a description of notes, warnings, and important messages.

Typographical Conventions and Syntax Notation

This manual 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 SonicMQ user interface elements. For example, "From the **File** menu, choose **Open**."

  Bold typeface is also used to highlight new terms when they are introduced in conceptual and overview sections.

- `Monospace typeface` is used to indicate text that might appear on a computer screen other than the names of SonicMQ user interface elements, including all of the following:

  – Code examples

  – Code that the user must enter

  – System output (such as responses, error messages, and so on)

  – Filenames and pathnames

  – Software component names, such as class and method names

  Essentially, `monospace typeface` indicates anything that the computer is "saying," or that must be entered into the computer in a language that the computer "understands."

  `Bold monospace typeface` is used to supply emphasis to text that would otherwise appear in `monospace typeface`.

  `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 symbol and font introduce a multi-step procedure:**

1. This is a first step.
   1.1 This is a step within a step.
2. This is a second step.

➤ **This symbol and font introduce a single-step procedure:**

☐ This symbol starts a single-step procedure.

This manual uses the following syntax notation conventions:

- Where command-line examples are provided, a backslash character (\) indicates line continuation. It should not be entered on the actual command line.
- 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 required items.
- Ellipses (. . .) indicate that you can choose one or more of the items.

Note, Important, and Warning Flags

This manual highlights special kinds of information by using shading, placing horizontal rules above and below the text, and using a flag in the left margin to indicate the kind of information.

Note A **Note** flag indicates information that complements the main text flow. Such information is needed to understand the concept or procedure being discussed.

Important An **Important** flag indicates information that must be acted upon within the given context for the procedure or task (or other) to be successfully completed.

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

Available Documentation

Table 1 lists the documentation supplied with SonicMQ. In addition to the documentation listed in this table, SonicMQ comes with sample files. All documentation is included with the SonicMQ media.

Table 1. The SonicMQ Documentation Set

Document	*Description*
SonicMQ Documentation Portal (`SonicMQ_Help.htm`)	Describes and links all SonicMQ online documentation components.
Getting Started with SonicMQ	Presents an introduction to the scope and concepts of the SonicMQ software and its packaging. Lists the features and benefits of SonicMQ in terms of its adherence to the Sun JMS specification and the extensions that make SonicMQ a richer, more useful messaging software.
SonicMQ Installation and Administration Guide	Describes configuration of various SonicMQ client types, clusters, and the message server and data stores. The administration chapters fully document server management using both the command-line interface and the graphical user interface administration tools. Covers security concepts and installation and administration of security features.
SonicMQ Programming Guide	Presents the SonicMQ sample applications and then shows how the programmer can enhance the samples, focusing on clients, connections, sessions, messages (including XML), transactions, and hierarchical topics.
SonicMQ Deployment Guide	The first part describes general deployment issues, including security. The second part concerns deployment issues for setting up dynamic routing for a B2B infrastructure.
SonicMQ API Reference	Contains information on the SonicMQ API that supplements the other manuals.
SonicMQ Product Update Bulletin	Describes enhancements to SonicMQ that are new with this release.
SonicMQ Release Notes	Provides late-breaking information and known issues.

Worldwide Technical Support

Progress Software's support staff maintains a wealth of information at **http://www.sonicmq.com** to assist you with resolving any technical problems that you encounter when installing or using SonicMQ Developer Edition.

From the SonicMQ home page, click on **Developer Exchange** to take advantage of resources for developers such as forums, downloads, tips, whitepapers, and code snippets.

For technical support for the SonicMQ Professional Developer Edition or the SonicMQ E-Business Edition, visit our TechSupport Direct Web page at **http://techweb.progress.com**. When contacting Technical Support, please provide the following information:

- The release version number and serial number of SonicMQ that you are using. This information is listed at the top of the Start Broker console window and might appear as follows:

  ```
  SonicMQ E-Business Edition [Serial Number 25677051]
  Release nnn Build Number nnn Protocol nnn
  ```

- Your first and last name.
- Your company name, if applicable.
- Phone and fax numbers for contacting you.
- Your e-mail address.
- The platform on which you are running SonicMQ, as well as any other environment information you think might be relevant.
- The Java Virtual Machine (JVM) you are using.

  To determine the JVM you are using, open a console window, go to the directory `SONICMQ_JRE` (default *`install-dir`*`\Java\bin`), and issue the command `.\jre -d`.

Table 2 provides information about Progress Software Corporation and its international offices.

Table 2. Progress Software International Offices

Locale, Office Name, and Address	*Contact Information*
North and Latin America: Progress Software Corporation 14 Oak Park Bedford, MA 01730 USA	**Pre-sales:** Telephone: 800 477 6473 ext. 4900 e-mail: `sonicmqpresales@progress.com` **Technical Support for Professional Developer Edition and E-Business Edition:** Telephone: 781 280 4999 Fax: 781 280 4543 e-mail: `support@progress.com`
Europe, the Middle East, Africa (EMEA): Progress Software Europe B.V. P.O. Box 8644 Schorpioenstraat 67 3067 GG Rotterdam THE NETHERLANDS	**Pre-sales:** e-mail: `sonicmqpresales-emea@progress.com` **Technical Support for Professional Developer Edition and E-Business Edition:** Telephone: 31 10 286 5222 Fax: 31 10 286 5225 e-mail: `emeasupport@progress.com`
Asia/Pacific: Progress Software Pty. Ltd. 1911 Malvern Road Malvern East, VIC Box 3145, AUSTRALIA	**Technical Support for Professional Developer Edition and E-Business Edition:** Telephone: 613 9885 0199 e-mail: `aussupport@melbourne.progress.com`

Chapter 1

Introducing Progress SonicMQ

Overview

Welcome to Progress® SonicMQ™, the leading E-business messaging infrastructure for the reliable, scalable and secure transport of business-critical data over the Internet.

Messaging is a system of asynchronous requests, reports, or events that are used by enterprise applications. It is an event-driven communications layer that lets applications—whether on the same system, the same network, or loosely connected through the Internet—transfer information as securely as needed and at whatever pace the one or many interacting systems can maintain.

SonicMQ is designed for use primarily with Java™ applications. It allows development and deployment of an efficient, secure messaging system that makes it possible for organizations to effectively (and reliably) communicate with various types of business systems over the Internet.

Evolution of Business Data Communications

Messages have always been a fundamental part of business. Whether scanning barcodes, buying from a catalog, or signing contractual documents, business relies on mutual acceptance of a set of information as the basis for an exchange of goods and wealth. Sound business practice has some basic principles:

- Every facet of every exchange must minimize time required to gain value as well certify that the goods and parties can deliver as intended.
- When all potential parties to a transaction can compete for the supporting services—financing, transport, warehousing, security, settlement—that comprise the total business venture, every transaction is more efficient.
- A portion of business time must be devoted to assess the business climate, knowing when to explore new channels and when to close existing ones, how to prepare for huge growth spurts and how to contain fixed costs.
- Merging and divesting businesses seems as easy as a handshake but the underlying accounting and auditing systems are often far less adaptive.
- Business policies must be codified to assure consistent application by all agents who act on behalf of the business.

Electronic commerce between businesses, **E-business**, maintains those principles yet changes the way businesses form and maintain relationships with other businesses.

In the new millennium, the Internet has become fundamental to every E-business initiative. Applications must exchange business data on the Internet. Much more than WebSite presence and consumer catalogs, businesses see the strategic importance of the emerging industry exchanges and supply chain integration. Two categories of businesses are forming: those that participate in and propagate sophisticated, adaptable global E-business exchanges and supply chains... and those that don't. One category will utterly dominate in less than a decade.

Before the emergence of the Internet, large businesses established extensive private networks to facilitate communication between business partners. This Electronic Data Interchange (EDI) required large startup costs and correspondingly large maintenance costs as each participant had to conform to the preset communication and application interfaces. EDI was—and still is—

reliable. But shifting business alliances and expanding world markets chafe at the rigidity and ponderous infrastructure of traditional EDI.

To open up from the tightly synchronized hardware and communications of EDI, Messaging-Oriented Middleware (MOM) provided the reliable data delivery mechanisms yet let the systems be loosely-coupled—not always operating at the same speed, sometimes disconnected, and not having the recipient synchronously lock until the communication was completed. MOMs coordinate message traffic between distributed applications, handling all network communications so that application developers need not be concerned with the underlying transport. Originally designed for the asynchronous exchange of information between mainframes in a corporate network, traditional MOM vendors focused on:

- Loose coupling of internal application processes
- Reliable exchange of information within the corporate network
- A high-level applications interface
- Quality of Service guarantees in Service Level Agreements
- Event-driven processing

Broadly accepted for internal application integration projects, companies are expanding their view of integration beyond the enterprise. To participate effectively in B2B commerce initiatives over the Internet, companies must adopt a global view of application integration that demands new capabilities from the underlying message transport.

Two primary, and fundamentally polar, models of messaging evolved:

- **Publishing messages** — Publication is much like broadcasting: A select set of information is sent to message servers. Active subscribers each get a copy of the message from the message server. The publishing model is appropriate for catalog updates, requests for bids or proposals, promotions, and financial updates.

  Some publish implementers describe a push-model where only known internet addresses or subnets are sent the message where routers might run daemons that monitor the subscriber base. The “serverless” publishing model is best for blasting streams of data such as complete stock activities, movies, television, and music.

- **Queuing messages** — Queues are destinations for messages that maintain each message as a unique object that will only be delivered to one recipient. Authorized receivers monitor relevant queues through filters, selecting to receive qualified messages. When a message is not acknowledged as received, it must return to the queue. When a message expires or becomes undeliverable, it can be transferred to special handling as well as trigger notifications. Sophisticated queue implementers, such as IBM® MQSeries®, enable messages to be routed through a message server to a queue on a remote server. The queuing paradigm with an extensible architecture enables E-business Messaging.

With the Internet emerging as a secure and reliable communications infrastructure, businesses can transfer data to partners and associates worldwide. Businesses can get data at their own pace and flow it into applications. But the Internet alone is a transport, usually doing little more than putting e-mail in front of people and web pages into browser windows.

An E-business Internet is possible when two other technologies are interwoven with it:

- **XML** — The data format of business, XML together with its associated stylesheets, parsers, object models, and data definitions, lets industries define data structures, making it easier for disparate applications to dynamically exchange information. XML files provide links to their validation structure so that the data can be mapped into the industry model and then into the application data formats.
- **Java Message Service (JMS)** — JMS is a standards-based Internet message service for Java that specifies the behaviors and mechanisms that enable both messaging paradigms to interact under the most sophisticated security and acknowledgement service levels. Among the advantages of JMS are:
  - Native support for Internet protocols and standards
  - Straightforward porting to new hardware platforms
  - Easy integration into heterogeneous networks
  - Standards-based technology in a family of business software solutions

SonicMQ: The Software for E-business Messaging

SonicMQ delivers everything a business needs to evaluate, design, deploy, and maintain all the most desirable features for high-performance and high-reliability E-business:

- Certified JMS implementation that provides a Queue Point-to-point messaging as well as a Publish/Subscribe domain.
- Extended client/server features that enable hierarchical security management.
- 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, and certificate management.
- Built-in XML messaging and integration with XML parsers.
- SonicMQ's Dynamic Routing Architecture (DRA) lets enterprises participate in global E-business exchanges through a single message server. 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 E-business integration by robustly providing five E-business essentials: scalability, availability, reliability, performance, and security.

Massive Scalability

With SonicMQ's Dynamic Routing Architecture, a B2B exchange can readily scale to thousands then tens of thousands of trading partners without requiring changes to the routing architecture or the trading applications. Effective connect-time load balancing and continued performance leadership allow SonicMQ to deliver the massive scalability required for E-business messaging.

NonStop Availability

The SonicMQ messaging infrastructure is up 7 days a week, 24 hours a day. SonicMQ's **clustering** allows multiple message servers to operate as a single routing node while managed from a centralized configuration server.When clustering techniques are implemented, a client application can detect dropped connections, dictate load balancing algorithms, and get failover support from the cluster. It's enhanced fault resilience that has no single point of failure.

Guaranteed Reliability

Messages can be guaranteed to persist when a message is sent to a queue with requests for persistence and capture in a dead message queue when undeliverable. Messages will be logged to SonicMQ's embedded database (or a database of the developer's choosing) to protect against system failure. When persistent messages are used with message server clusters, clients are guaranteed to receive all messages pertaining to their chosen destination. Persistent messaging ensures that messages are never lost due to network or hardware failure.

Mobile users, although connected to the network frequently, need not be concerned that they missed out on messages published when they were unable to receive them. In a sometimes-connected world, a subscriber to a topic can express to a message server that selected messages should be retained under a durable subscription.

Very High Performance

Benchmark results confirm SonicMQ's superior performance. Metrics show that a SonicMQ message server can support up to 2,000 concurrent connections with a 10 MB 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.

End-to-end Security

Security is crucial in a global trading environment. And SonicMQ provides security levels required by these systems:

- SonicMQ message servers, clusters, and dynamic routing nodes manage access control to servers and destinations. A challenge-and-response protocol is implemented for user authentication.
- Message security can use certificate-based mutual authentication through RSA's SSL implementation to get full server-server and client-server security including certificate identity management and certificate generation.
- A message payload can be encrypted with an associated digest so that both message privacy and integrity are certifiable. (56-bit MD5/DES).
- SonicMQ's SSL protocol implementation supports up to 128-bit encryption.
- SonicMQ provides transport protocols that leverage popular Internet security mechanisms for messages and transport through multiple firewalls using flexible HTTP/HTTPS tunneling with forward and reverse proxy support.
- Message server-to-client authentication through PKI (public key) is also supported.

Adherent to Connectivity and Application Standards

SonicMQ has taken a leading role in its support for:

- **JMS Specification** — The Java Message Service v1.0.2 specification is fundamental to the architecture and interfaces of SonicMQ.
- **JNDI** — The Java Naming and Directory Interface is used throughout SonicMQ.

- **Internet Protocols** — SonicMQ supports standard Internet Protocols including Secure Socket Layers (SSL), HTTP tunneling, encryption, TCP, and security.
- **Internet Security** — Firewall tunneling, SSL, and digital certificates are supported with SonicMQ.
- **XML Message Type** — This emerging standard for distributed application communication is built into the SonicMQ XML message type and DOM-capable client.
- **100% Java** — SonicMQ's pure Java implementation is consistent so that development tools, platforms, and other applications that maintain this rigorous purity are not only easier to work with and easier to integrate—applications are easy to port to other platforms.

Bridges for Extended Data Integration

- **ActiveX/COM Client API**
- **Thin C-Client**
- **Bridges** — SonicMQ offers bridges to native MQSeries implementations, JMS-compliant messaging systems, and e-mail services under SMTP, IMAP, and POP3.

Concepts

SonicMQ implements standardized messaging concepts, providing the developer an easy-to-use set of interfaces and administrative tools. The Application Programming Interfaces comply with the standards in the Sun Java Message Service specification v1.0.2.

The following information describes the messaging concepts from the general toplogy to connectivity to the alternative messaging behaviors to the attributes and characteristics that enable secure, reliable messages.

Direct Application Integration

When messaging is not implemented and distributed applications are integrated, as shown in Figure 1, every client application maintains a connection to every other client application.

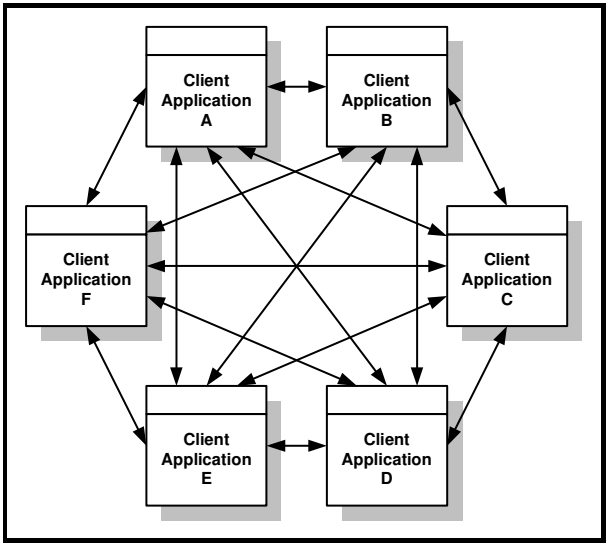


Figure 1. Distributed Application Structure

Once applications are linked together, they might be secure, fast, and reliable. But any advantages of direct peer-to-peer communications are compromised by several limitations:

- Bringing large integrated systems online is often so expensive and time consuming that systems threaten to become obsolete as soon as they start to amortize their costs.
- Outsourcing business processes is resisted because re-architecting communications and data transfer protocols is too costly.
- Modifying application software with hard-coded communication mechanisms is tedious and error prone.

Message Server Distributed Application Structure

The message server architecture, shown in Figure 2, considers every entity in its local topology to be a client except the **message server**—the entity to which every client connects, thereby providing connection services to every other client.

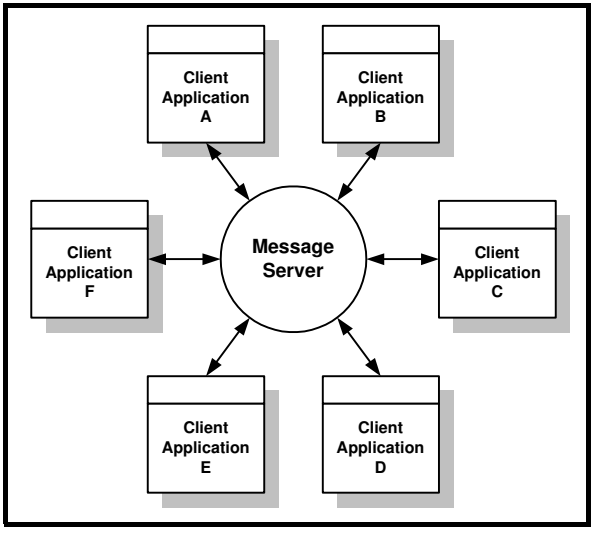


Figure 2. Message Server Structure

Aside from the savings in connections (this topology needs only *n* connections for *n* clients), the messaging system is far more efficient for several reasons:

- The message server manages connection logistics and protocols, providing the latest connections to clients.
- Security is maintained at the central hub, the message server.

The message server structure can expand to increase reliability and performance, yet the clients need only know how to connect to the appropriate port on their preferred message server system.

The components that are needed to implement and manage a JMS application are supplied by the **JMS provider**. This includes the JMS Client API and the SonicMQ Client Run Time accessed from within the client application, the communications layer between the client and the message server, and the message server architecture. The message server listens on a port on its host

system to provide services to its clients. The message server architecture and services are central to SonicMQ's messaging security, efficiency, scalability, and performance.

The SonicMQ message server does the following:

- Manages the persistent data store. Types of data that are persisted include:
  - **Message Store** — Stores and retrieves connection factories, queues, queued messages, and messages held for durable subscribers and their subscriptions.
  - **Security Database** — When security is enabled, manages the Access Control Lists for authentication of users and permission for users to read and write to queues and topics.
  - **Configuration Server** — When server cluster activities are enabled, maintains the configuration so that multiple message servers can interact, enabling robust and secure cluster topologies.
- Provides simultaneous multi-protocol support for TCP and HTTP connections, HTTP tunneling, and Secure Socket Layers (SSL).
- Logs activities.

Administration

The administration tools—the command-line Admin tool, the GUI Explorer, and the methods in the Management API—provide secure access to the configuration database, the security database, and the message store to control:

- Queues and topics including the security and Quality of Protection settings.
- Routing users, connections, and global destinations.
- Message server and cluster configurations.
- Users, groups, and access control lists.
- Administrative notices and monitoring of the dead message queue.
- Metrics and performance.

Persistent Data Stores

A data source bound to a message server provides the repository for all the administered data, the security database, and the message store.

See the *SonicMQ Installation and Administration Guide* for more information about persistent data stores.

Clustered Message Servers

Message servers can be linked together to provide a unified set of services for clients. With cross-server awareness (**advertising**) of queue destinations, clusters provide failover and load balancing mechanisms yet are viewed in the messaging topology as a single **routing node**.

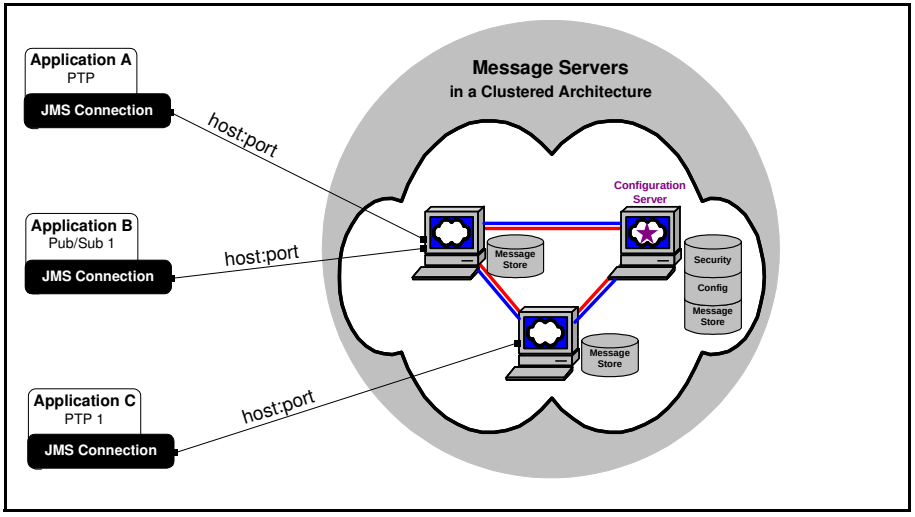


Figure 3. Message Servers in a Clustered Architecture

In a clustered architecture, SonicMQ also manages the persistent data store for the configuration server . The configuration lets multiple message servers interact, enabling robust and secure cluster topologies.

Client Applications

Client software sends and receives messages in a network of message traffic. A single client application could perform three distinct functions:

- Interface with a business application to maintain business records.
- Assess the content of a message to determine where some or all of the message content should be forwarded.
- Examine routing information to forward unopened messages.

Agent Applications

Systems that are linked to record-keeping systems are normally the starting point and endpoint in a message lifecycle. Realtime devices and accounting document lifecycles create messages—counters, metrics, and switches on devices, and accounting purchase orders, bids, invoices, backorders, production scheduling, packing orders, bills of lading, shipping, duty fees, quality control, receipts, and inventory changes—that are moved into the messaging stream by agent applications. Correspondingly, receivers gather appropriate messages to funnel into their controlling application, possibly returning an audit trail identifier when the accounting records are updated.

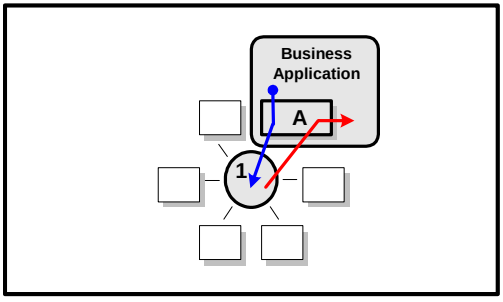


Figure 4. Agent Application

Figure 4 shows a Business Application using agent application **A** to produce and consume messages at destinations on message server **1**.

Transformation Applications

A technique in messaging that has great value is a **transformation application**. Transformers watch for messages that their embedded business logic might be able to break into pieces appropriate for several messaging channels. By exposing the granularity of the message, each element of a message might proceed on to a different path.

For example, a construction requirements message—when formatted in industry-approved XML tagged text—might be broken out into architectural phases so that appropriate subcontractor bids can be sent as separate messages.

As Figure 5 illustrates, an application receives a message from a message server and breaks up the message through connection to other message servers, effectively consuming and reproducing the message. But the services that were associated with the original message could be seriously compromised in this step. And the complete content of the original message could be lost. While transformations are a nifty way to split messages across business realms, they are inappropriate for any concerted effort to construct a complete E-business.

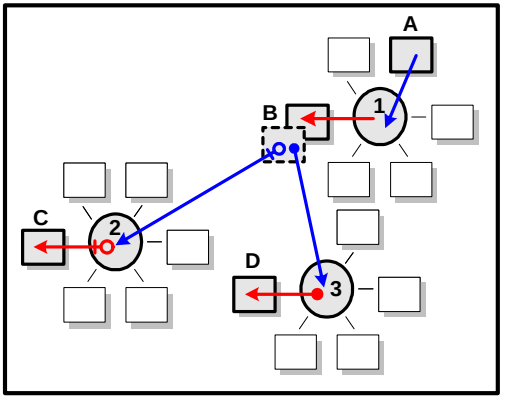


Figure 5. Transformation Application

In Figure 5, a message is transformed from its sender to its ultimate recipients as follows:

- Application **A** sends a message to a queue on its local message server, server **1**.
- Application **B** receives the message from server **1**, examines it and determines that it can send part of the message to server **2** and the other remainder to server **3**. Application **B** then acknowledges the receipt of the original message from message server **1**.
- Application **C** receives the message from the queue on message server **2**.
- Application **D** receives the message from the queue on message server **3**.

Routing Applications

When an application works with messages for the sole purpose of forwarding the message without touching its content and without changing the intended service levels, that application is a **routing application**.

Every message has information exposed in its metadata—the message header fields, and the properties—that enable a routing application to choose messages by defining qualified messages that it will receive in a message selector string. When a message is received by the routing application, it clones the message, looks up the data that tells it what the next destination should be, updates the message's destination, sends out the clone and then acknowledges and discards the original message.

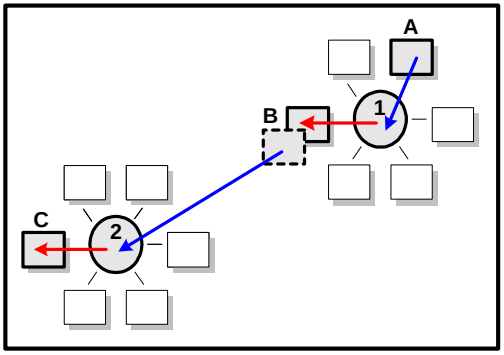


Figure 6. Routing Application

In Figure 6, the transformation at application **B** is a transformation of only routing information in the message header. The message is routed from its sender to its ultimate recipients as follows:

- Application **A** sends a message to a queue on its local message server, server **1**.
- Application **B** receives the message from server **1** because **B**'s selector knows that the message can be forwarded to server **2**. Application **B** then acknowledges the receipt of the original message from message server **1**.
- Application **C** receives the message from the queue on message server **2**.

Dynamic Routing Applications

SonicMQ's Dynamic Routing Architecture enables messages to be routed across nodes so that the messaging flow from the sender to the ultimate recipient is more efficient, and enforceable by the administrator of the routing node. In Figure 7, message server **1** has a **routing table**—a list of servers and queues that the sender can request and the server can handle—that enables the originator of the message to present the message to the server whose tasks are to first validate that the target queue on message server **2** is a registered destination, and then to "store and forward" the message to server **2** on behalf of the sender.

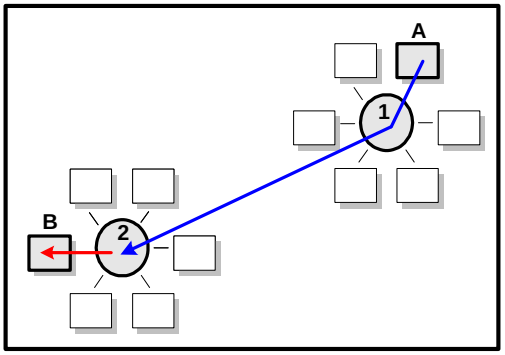


Figure 7. Dynamic Routing's Store-and-forward mechanism

In Figure 7, a message is addressed by its sender directly to a queue on a message server that is accessed by its recipient, as follows:

- Application **A** sends a message to its local message server, **1** with a destination name preceded by "**2::**".
- Message server **1** consults its routing table and verifies that the intended queue on message server **2** is an acceptable destination so it stores the message in its database (if requested to do so) then acknowledges the sender. Message server **1** forwards the message to the queue on server **2.**
- Application **B** receives the message from the queue on message server **2**.

Trading Partners and Portals

The Dynamic Routing Architecture enables sophisticated topologies for E-business that are reliable and manageable. A trading partner and portal structure provides a central business hub that manages data flow through **portal applications** that can transfer a message in a queue on the portal message server to the message server of a trading partner.

In this architecture, the portal brokers communications between trading partners without the partners having to be concerned about each other.

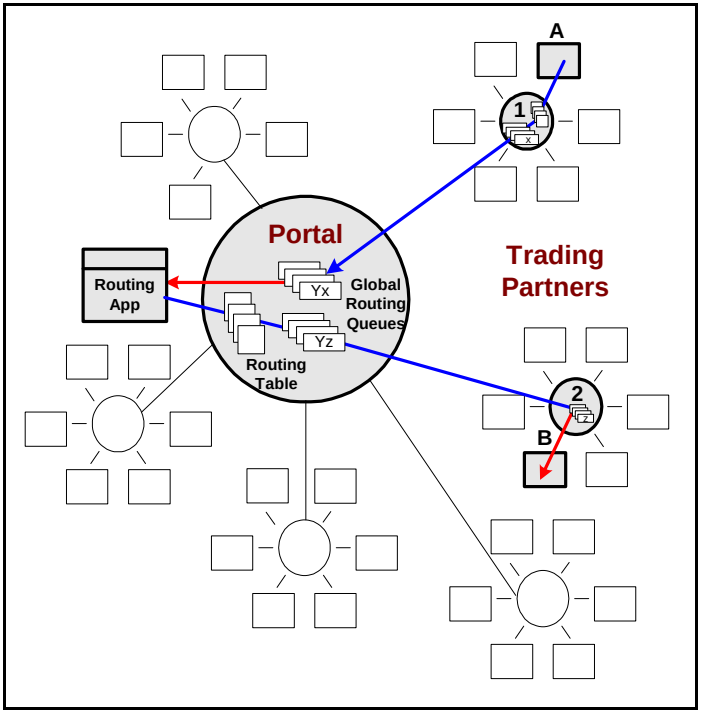


Figure 8. Portal and Trading Partners

In Figure 8, a message is addressed by its sender directly to a queue on the portal where one of its trading partners will be the recipient, as follows:

- Application **A** sends a message to its local message server, server **1** with a destination name "**Portal::Yx**".

- Message server **1** consults its routing table and verifies that the **Yx** queue on the **Portal** is an acceptable destination so it stores the message in its database (if requested to do so) then releases the sender. Message server **1** then forwards the message to the **Yx** queue on the **Portal.**
- A **Routing App** is authorized to work directly on the portal monitors. It listens on the **Yx** queue, selecting messages with attributes described in the message's metadata—the system defined messaging properties as well as special properties known to the trading partners. It receives the message.
- The routing app preserves the message content and intended services and determines from business rules where to forward the message. The message is cloned, its destination restated as "**2::z**" and the message is sent.
- The **Portal** consults its routing table and verifies that the **Yz** queue on the message server **2** is an acceptable destination so it stores the message in its database (if requested to do so) then releases the sender. The **Portal** then forwards the message to the **z** queue on message server **2.**
- or The **Portal** authenticates connections by the trading partners. The **Portal** looks at message attributes and then examines business relationships between partners. Business logic is applied to forwarding messages to appropriate partners and services of the **Portal**.
- Application **B** receives the message from the **z** queue on message server **2**.

Advantages of Trading Partners and Portals

Trading partners and portals use the Dynamic Routing Architecture (DRA) to ensure reliable messaging between message servers that can be independently configured and administered. With the DRA, thousands of companies can work together through portals. Trading partners become clients to the portal message server.

In a portal-and trading-partner architecture, basic messaging services enable:

- **Local Management** — Each trading partner can maintain its own local message server and control access to that local server.
- **Disconnected Service** — Each trading partner has a Store-and-Forward feature so that disconnection from the portal is not an issue.
- **Complete Security** — Each trading partner can implement security firewalls, secured destinations and rigorous access control. Similarly, the portal, as a hub, can maintain a firewall, secured destinations and rigorous access control.
- **Routing** — The **Portal** acts as a router in the marketplace by determining how to send messages between partners.
- **Scalability** — The whole messaging infrastructure must be able to expand to accommodate the growth of E-business exchanges while still providing service to many more users and partners.
- **Management and tuning** — Several features of SonicMQ make deployments of E-business exchanges easier to monitor and adjust:
  - **Flow control** — Clients and servers can choose to how to react when message throughput gets clogged.
  - **Load balancing** — Several servers can be listed as alternative connection routes to balance individual server outages or overloads.
  - **Failover mechanisms** — Clustered server mechanisms can re-route server traffic during outages.
  - **Dead message queues** — Messages can be set to transfer to a special persistent queue if the message expires or cannot reach its destination.
  - **Management notifications** — Notifications to administrators when messages are undeliverable can highlight emerging problems.
  - **Performance tools** — Special software tools that assess flows and traffic can assist administrators in fine tuning system performance.
  - **Instrumentation** — Metrics on messaging activity provide a visual log of messaging system activity.
  - **Management API** — Many administrator functions can, under appropriate authorization, be coded into management applications.

How SonicMQ Works

When you understand the mechanisms and behaviors of SonicMQ applications, you are knowledgeable about the Java Message Service 1.0.2 specification. Several concepts in SonicMQ are presented here that are extensions of the JMS specification—such as the XML message type, prefetch, single message acknowledgement, flow control, and load balancing—that can make better, more efficient implementations. There are a few features of the JMS specifications that are optional and not yet implemented in SonicMQ, such as XA transactions and server session pools.

The rest of this chapter shows you how SonicMQ works.

Connections and Sessions

A SonicMQ application starts by accessing a ConnectionFactory to create a **connection** that binds the client to the message server. ConnectionFactories are **administered objects**, objects with connection configuration parameters that have been defined by an administrator.

Within a connection, one or more **sessions** can be created. Each session establishes a context for one thread where messages might be sent or received. Figure 9 shows a client application where one connection has been made through which one session has been established. The client application uses programmatic interfaces to the JMS Client API that are executed through the SonicMQ client run time on the session.

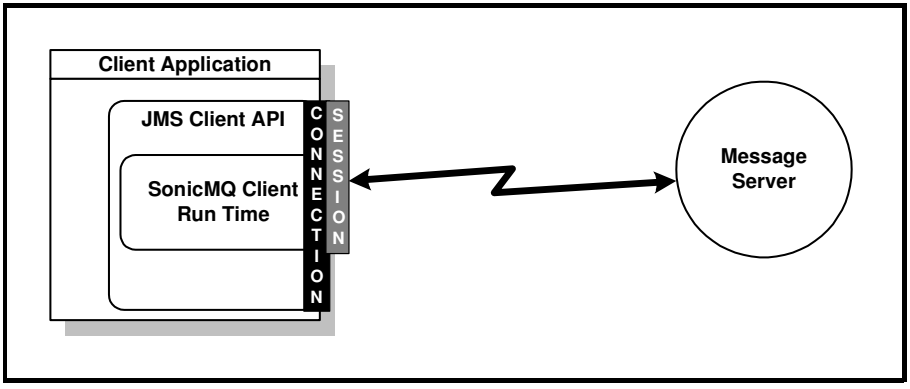


Figure 9. JMS Session on a Connection

Concurrency

A client can create multiple sessions within a connection to the message server, each independently sending and receiving messages. Sessions execute in parallel so that multiple threads are active concurrently, letting developers optimize thread usage for co-dependent activities.

For example, a connection might have two sessions where one has a registered listener for receiving messages while the other is dedicated to listening to standard input and sending messages, thus removing the possibility of concurrent activities on a single thread.

A single session can be used for sending and receiving if careful programming ensures that more than one thread will not have access to session at one time.

Session Type

Each session is established with a declared intention for acknowledgement of messages. A session is can be transacted or non-transacted, in which case one of several modes can be specified for acknowledging messages.

- **Transacted Session** — A SonicMQ session can be specified as **transacted**. The technology of transaction processing significantly reduces the effort required to build applications by allowing applications to combine a group of one or more messages with publisher-to-message server **ACID** properties:
  - **Atomic** — Either all the messages are delivered or all are ignored.
  - **Consistent** — Business rules ensure the receiver does the right thing.
  - **Isolated** — Transacted messages will be delivered serially.
  - **Durable** — The messages are all persistent.

  When a transaction **commits**, its atomic unit of input is acknowledged and its series of messages is sent. If a transaction **rolls back**, its produced messages (if any) are destroyed. The completion of a session's current transaction automatically begins the next transaction.

  A transacted session is a good example of a session function that can take advantage of sending and receiving in a single session. When a transacted session commits, all the sent messages in the session are released on the server destinations and all the received messages are acknowledged.

- **Non-transacted Session Acknowledgement Mode** — When a nontransacted session is created, the client application can set the type of acknowledgement it expects when messages are delivered:
  - **Auto** — The session automatically acknowledges the client's receipt of a message when the session has successfully returned from a call to receive, or the **MessageListener**, called to process messages, successfully returns.
  - **Client** — The acknowledgement of a received message automatically acknowledges the receipt of all messages that have been delivered to a consumer by its session. A session can recover so that it restarts with its first unacknowledged message.
  - **Single Message** — Explicit acknowledgement of one message.
  - **Dups_OK** — The session “lazily” acknowledges the delivery of messages to consumers, possibly allowing some duplicate messages after a system outage is experienced. In systems such as market quote streamers, duplicates are acceptable if faster system recovery is the trade-off.

While acknowledgement sets standards for message delivery, there is no reply to the sender. If a reply to the sender is required, a message attribute is reserved for the request. The requestor can also append a correlation identifier in the message’s header that will ensure that the reply matches its request.

Producers and Consumers

The traffic on a session thread to the message server, as shown in Figure 10, could consist of a message producer delivering a message to its message server or a message server delivering a message to an application that will consume it.

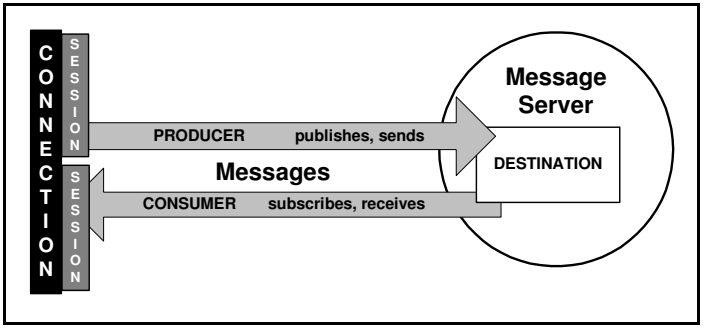


Figure 10. Producers and Consumers

The **producer** of the message packages and encrypts the message body, identifies the service level and protection for the outbound message, and then sends the message to its destination (a specified location in the message server's realm).

The **consumer** of a message binds to a destination to receive a message and then implements the message's delivery method:

- **Synchronous delivery** — The client requests the next message using a receive method that polls the session's MessageConsumer for a destination. Synchronous delivery waits for its reply, the fundamental problem with traditional system interconnections. The connection could be blocked indefinitely.
- **Asynchronous delivery** — The client registers a MessageListener. As messages arrive, SonicMQ calls the listener's `onMessage` method. With an asynchronous connection, if the connection is in any way impaired, messages that are guaranteed will wait—in the message server's database—for a connection to be re-established. The producer can determine how long a message will wait for a consumer so that backlogs can be avoided.

Delivery Mode

When a message is sent to a destination, additional effort can be applied to assuring that the message will always be stored in a local log or data store before it is acknowledged as being transferred. This **`PERSISTENT`** delivery mode

ensures that a message is safe from normal system outages and interruptions—whether in its initial transfer to the message server, in durable subscriptions or queues on the server, and even in transfers to other message servers and queues.

When messages have a **NON-PERSISTENT** delivery mode, the message throughput is faster but messages are volatile—they might not be recoverable from an outage.

A message can also be **NON-PERSISTENT_ASYNC**, an even faster delivery mode where the producer does not get blocked on its thread to the server when a message is produced, effectively choosing no acknowledgement at all. As a result, a strong flow of messages can be pushed to the server without delay.

Destinations

Destinations are the delivery labels in messaging. Rather than the place where the message is ultimately delivered, a destination is the commonly understood staging area for the message.

The overview of SonicMQ distinguishes the two JMS messaging domains:

- **Publish and Subscribe** (**Pub/Sub)** — Produces messages to a **topic**. Prospective consumers of messages addressed to a topic simply subscribe to the topic. While a message can have many subscribers (one-to-many), the producer does not know how many subscribers, if any, exist for a topic.
- **Point-to-point (PTP)** — Produces messages to a named **queue**, placing new messages at the back of the queue. Prospective consumers of messages addressed to a queue can either receive the frontmost message (thereby removing it from the queue) or browse through all the messages in the queue, causing no changes. While several clients can access a queue, a message is received by only one (one-to-one communication).

These messaging domains are substantially the same (except for some semantic and syntactic differences) in terms of connection and session management, message structure (types, headers, and properties), delivery mode options, and delivery methods (blocking or asynchronous).

Publish and Subscribe Messaging: Broadcast the Message

The Publish and Subscribe messaging model involves a broadcast of messages: Many subscribers get precisely the same message, as shown in Figure 11.

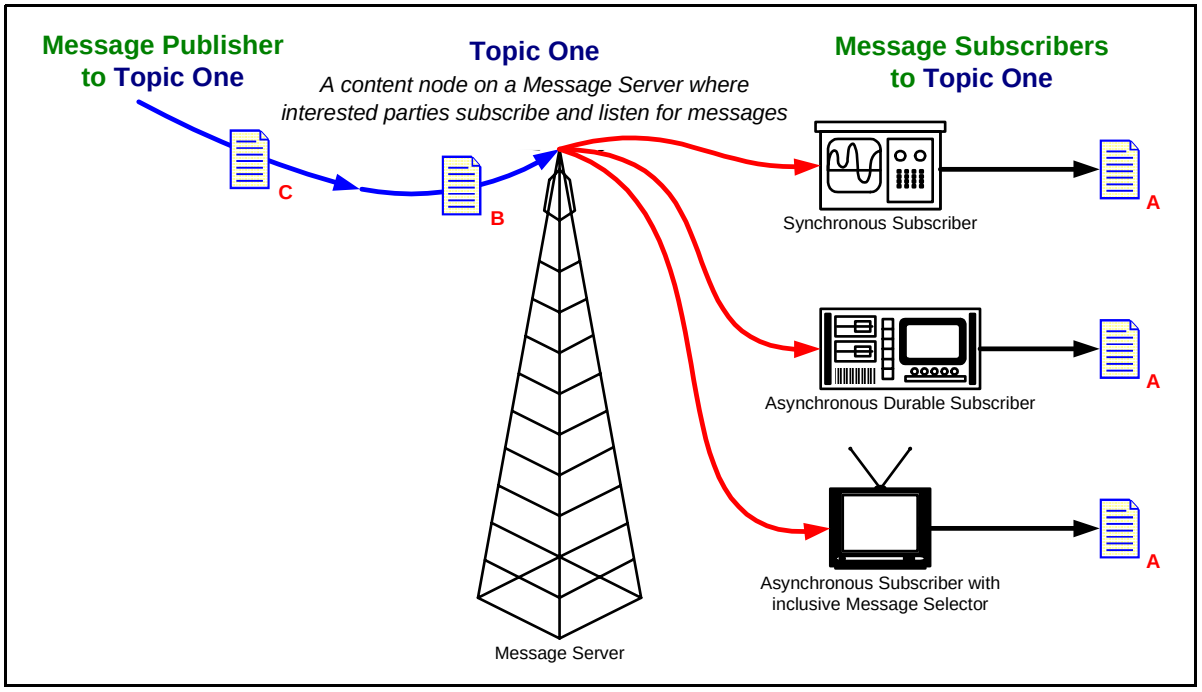


Figure 11. Concept of Publish and Subscribe Messaging Topics

The Publish and Subscribe diagram shows three subscribers who have each received message **A** and are about to receive message **B** then message **C**:

- A **synchronous subscriber** waits for a message - for a specified time or forever - and then blocks to receive again after processing a message.
- A **durable subscriber** recorded an interest in receiving messages from the message server 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.
- An **asynchronous subscriber** has set up a message listener. When a message arrives, the `onMessage` method delivers the message to the

consumer process. In the diagram, this subscriber is noted to have a **message selector**. The subscriber provided a string in SQL syntax as a parameter when the subscriber was created. If the selection criteria are not met, the application's consumer does not take delivery of that message.

In the Pub/Sub model, a topic might have anywhere from no subscribers to millions of subscribers. There is no statistical feedback to the publisher to indicate demand but there is also no burden on the publisher when the number of subscribers increases sharply.

How Publish and Subscribe Works

In the Pub/Sub paradigm, a producer is known as a **publisher** and a consumer is a **subscriber**. A publisher initiates a message by sending an instance of a message type object that has the appropriate body content as a payload and header and property data exposed to aid in delivery and tracking. The publisher declares the quality of service—delivery mode, time-to-live, and priority—as well as whether a reply is requested from the consumer. The message server returns a message identifier, and immediately distributes the message to all consumers of that topic. If the delivery mode is **`PERSISTENT`**, the message is placed in the message server's log or message store before starting delivery.

In Figure 12, the Pub/Sub session thread has publishers producing messages to topics maintained by the message server and subscribers consuming messages that the message server delivered from topics where the session is subscribed.

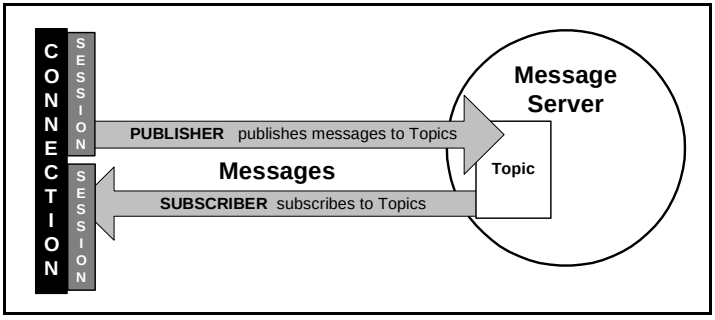


Figure 12. Publishing and Subscribing on a Topic Connection

After delivery of the message, the message server checks to see if any of the subscribers to the topic are **durable subscribers**—subscribers who expressed

a durable interest in the message's topic. If, after initial delivery, any durable subscribers did not acknowledge delivery, the message is retained until the expiration time in anticipation that a durable subscriber will connect to the message server and accept delivery. The **expiration time** is calculated from the time-to-live beyond the time of publication. A message can be set to live forever but will still be discarded as soon as delivery to all current subscribers and all durable subscribers is complete.

Asynchronous subscribers use a message Event Listener that will deliver the message to the subscriber's application for interpretation—de-encryption, parsing, and passing to the message consumer's methods.

Subscribers can filter the messages they receive by qualifying their subscriptions with **message selectors** that will evaluate message headers and properties (but not the content) with expression strings created with a subset of SQL-92 semantics.

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

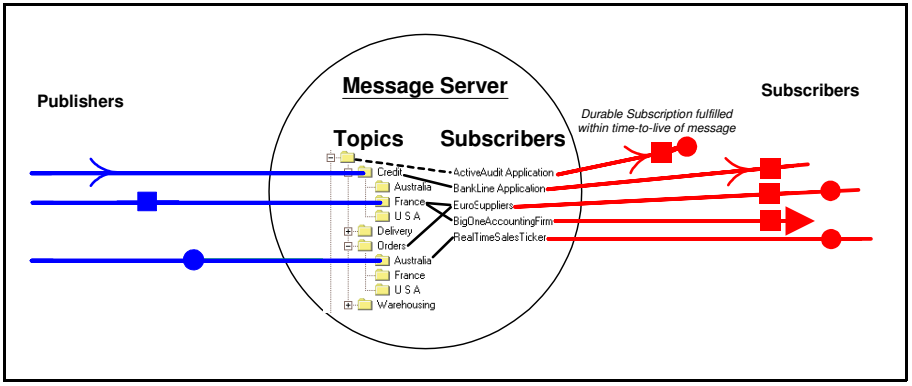


Figure 13. Publishing Messages to Topics for Subscribers

Figure 13 describes how publishers send messages to topics and how the messages are routed to normal and durable subscribers:

- Publishers are sending messages to specific topics.
- The message server is keeping track of messages and security for both active and durable subscribers to topics and topic name spaces.
- As soon as messages are published they are distributed to the subscribers. Durable subscribers who were inactive get messages when they reconnect within a specified time.

Point-to-point Messaging: There is Only One Message

The Point-to-point messaging model ensures a singular delivery of a unique message. A message is intended to be delivered to a single consumer.

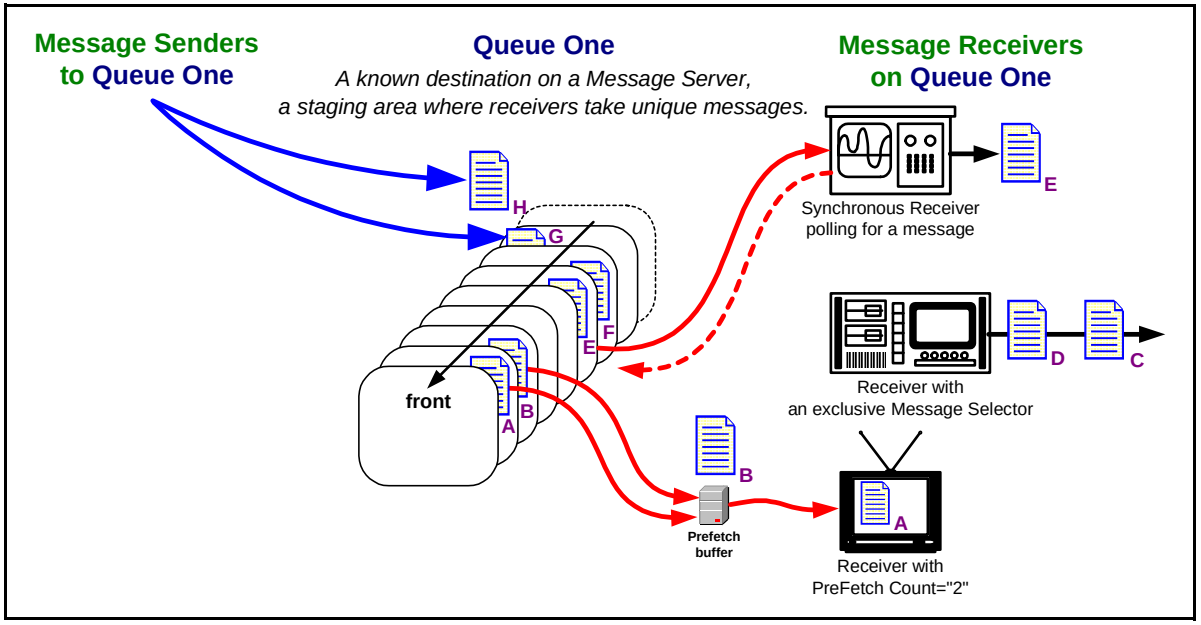


Figure 14. Concept of Point-to-point Messaging Queues

In Figure 14, the message senders send new messages to **QueueOne**, a destination on the message server. The message server, unless advised that there is a request for priority treatment, places new messages at the back of the queue.

The messages in this illustration are being removed from the queue by three receivers:

- A **synchronous** receiver waits for a message—for a specified time or forever—and then blocks to receive again after processing a message. The queue state above indicated that message **E** was the frontmost message because messages **A** and **B**—while still in the queue—are awaiting acknowledgement from another receiver.
- A receiver browsing with a **Message Selector** reviews qualified messages on the queue to determine if any of them are messages that it wants to process. In this example, the receiver selected and acknowledged messages **C** and **D**. Assuming message **F** does not meet its criteria, this receiver perceives a momentarily empty queue.

  By also using a **QueueBrowser**, this receiver can scan through a queue capturing the ever-changing queue image as it progresses. The QueueBrowser mechanism can also let the sender peruse the queue to see how message traffic is moving.
- A receiver with a **PreFetch** count of **2** took messages **A** and **B** off the queue. Message **B** is held aside while message **A** is processed. When message **B** enters processing, the threshold trigger compels the receiver to draw off two more messages. The message server can keep track of these messages in process and, unless acknowledgement is received, the messages will be reinstated into the queue.

In the PTP model, each message is a unique item. But messages can disappear without ever getting delivered. A message will simply expire when its time to live has passed.

A message could, in more complex server exchange architectures discussed later in this chapter, get stuck in a position where the sender is given confidence that the message server accepted the message but other message servers, routes, or destinations could create a state of doubt about final delivery. An application can send messages with properties that express an intent to have the

dead message preserved or at least tell the administrator the reason it was undelivered.

This concept, a **Dead Message Queue**, can work in concert with other Quality of Service features to provide **guaranteed persistence** in the always-active yet sometimes-disconnected networking environment of modern computing.

How Point-to-Point Communication Works

In the PTP paradigm, a producer is known as a **sender** and a consumer is a **receiver**. Point-to-point messaging queues are explicitly created by the administrator to allow for sequential caching of messages for their receiver. While only one receiver will consume the latest message, several receivers could be reading from the queue, taking turns consuming the latest message. A PTP message producer sets headings, properties, and body content in much the same way as Pub/Sub producers.

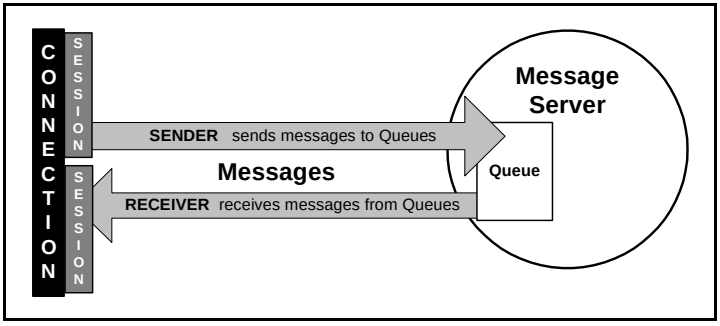


Figure 15. Sending and Receiving on a Queue Connection

In Figure 15, the PTP session thread has senders producing messages to destinations maintained by the message server, and receivers consuming messages that the message server forwarded from queues where the session is waiting to receive messages.

The receiver of a queued message, however, notices a remarkable difference between the Pub/Sub and PTP paradigms. In PTP:

- **The first message received is the first message delivered.** This FIFO technique makes the second through **n**th messages endure until that first message is consumed. Even when no clients express interest in

receiving messages from a queue, messages wait for a receiver until the message expires. When a message's delivery mode is set to **PERSISTENT**, the message is stored so that even a message server shutdown will not put it at risk.

- There is **only one message consumer** for a given message. Many prospective receivers can balance the load, but only one takes delivery of the message.
- When the message is acknowledged as delivered, it is removed from the queue permanently. No one else sees it and no one else gets it.

Authorized users of the **Queue Browser**, a mechanism that examines queues, can take advantage of durable sequential queues, to allow scanning messages without destroying them.

In Figure 16, several FIFO queues are shown that might exist in a message server's queue session management. The queues are shown to have different depths, portraying that the stack of messages persists until receivers take messages off the queue as fast as they are added. The queues also show a receiver taking a message.

The receiver could be one of many receivers who are standing by to receive the topmost message. On the middle queue (**BQ**) in Figure 16, multiple receivers are allowed but only one receiver gets the message: A queued message gets delivered only once.

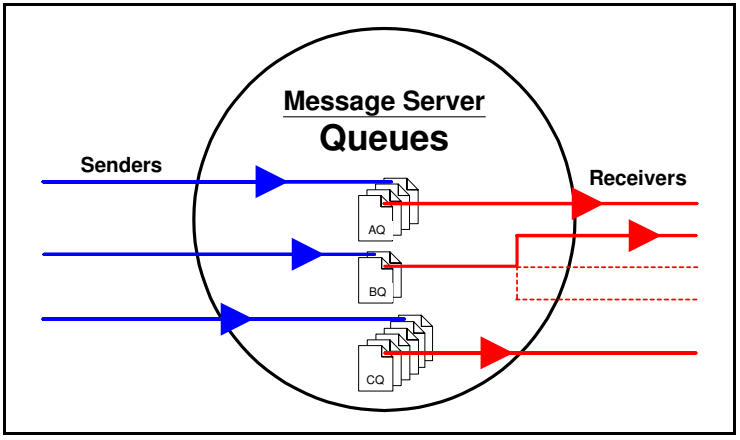


Figure 16. Sending Messages to Queues for Receivers

Queues are the preferred load-balancing paradigm when many diverse systems can share processing operations mandated by heavy trading activities, credit card charges, online shopping carts, auctions, reservations, and ticketing.

Messages

A SonicMQ message is fully compliant with the JMS specification of a message with all attributes implemented plus a few important extensions. The message types are:

- **Message** — Basic message where no body is required.
- **TextMessage —** A standard `java.lang.String`.
- **XMLMessage —** A SonicMQ-specific derivation of the `TextMessage` 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 the values are Java primitives.
- **BytesMessage —** Stream of uninterpreted bytes.

Structure

A message is comprised of a set of header fields, a set of extensible property fields, and—in most cases—the body of the message:

- **Header** — A message header contains name-value pairs used by both producers and consumers to identify and route messages. JMS-standard header fields are `JMSCorrelationID`, `JMSDestination`, `JMSDeliveryMode`, `JMSMessageID`, `JMSTimestamp`, `JMSReplyTo`, `JMSRedelivered`, `JMSType`, `JMSExpiration`, and `JMSPriority`.
- **Properties** — Message 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 receiver into other acceptable data types. SonicMQ defines a few properties that it uses to control and manage undeliverable messages.

- **Body** — The message body is a set of bytes interpreted as its message type. SonicMQ provides the five message types defined by JMS and extends the `TextMessage` type to implement the `XMLMessage` type. The message body is actually optional: You can send a body-less message where the header and property values contain all intended information.

Request/Reply

A message producer can declare that it wants more than just acknowledgement from the message server that it received the message. The producer can set a flag indicating that it would like any consumer of the message to provide an explicit acknowledgement directly to the producer at a stated destination.

To confirm delivery to its consumer, the message field `JMSReplyTo` is used as an indicator that a response is anticipated from the consumer and where that response message should be sent. The action is in effect a reversal of roles whereby the consumer is asked to produce the reply message and the original message producer waits at the **temporary destination** to consume the return message. While this certifies delivery, in its simplest form it is synchronous and therefore a blocking action for the original producer. However, there are mechanisms that enable asynchronous requests and replies.

Quality of Service

Each client application can independently configure the type of message delivery for a particular destination. Quality of Service is supported by session options and producer parameters:

- **Acknowledgement Mode** — The session option determines whether the acknowledgment of communications between the client and the server are controlled by the client, the server, or are simply done with reasonable efforts.
- **Message Expiration** — Messages can be sent with a specific life span to ensure that clients do not receive out-of-date information. When a message expires, it is dropped from the queue or from the unfulfilled durable subscriptions.
- **Delivery Mode** — A **persistent** message is stored in the message server's logs and repository for later delivery to potentially disconnected users. This action provides a higher quality of service yet produces a

corresponding decrease in performance. A persistent message will survive a system disconnection or unexpected restart. Persistence is maintained as a message moves across servers as it is routed.

- **Guaranteed Persistence** — To make sure that a message will be delivered or exist on a queue, a message can be flagged for transfer to the system **dead message queue** when it expires or gets into an undeliverable situation. On the DMQ, a message never expires and can only be discarded by explicit action by the administrator.
- **Priority** — Messages can be sent with a priority value that encourages the message server to position that message ahead of other messages in the same queue or topic.
- **Redelivery** — The message server can make repeated attempts to redeliver messages to each client that has not acknowledged receipt.

In addition, SonicMQ has features that enable application design patterns to monitor the client server connection and take efforts to reconnect on a different server when a server loses a connection.

Quality of Protection

Even when assured of persistence, message integrity can be compromised such that unauthorized entities can read or change the message. SonicMQ provides features that enable a message to have an enhanced Quality of Protection (QoP) level:

- **Integrity** — The contents of the message cannot be altered without the recipient (and possibly the sender) being informed of the changes.
- **Privacy** — The content of any individual message cannot be viewed by anyone other than the intended recipient. When SonicMQ applies **Privacy**, it includes **Integrity** as well.

Security

SonicMQ allows an administrator to associate a security policy with every destination, determining which entities can produce or consume messages to topics and queues. SonicMQ's Access Control Lists (ACLs) manage the authentication of usernames and the events and destinations where a user is authorized to perform actions.

Chapter 2

Setting Up SonicMQ Developer Edition

Contrasting Developer Edition and Other Editions

Consult your Progress representative to learn more about the SonicMQ products and appropriate DBMS vendors for the persistent message store and security database.

There are three editions of SonicMQ available:

- **SonicMQ Developer Edition** is fully functional as a basic message server and is designed for easy installation. It is intended to let you get acquainted with SonicMQ and to explore its features and performance by building and testing a fully-functioning messaging system. The Developer Edition server is constrained to accepting a modest number of clients, 100, but every client connection must initiate from the same specified IP address. The Developer Edition clients and the associated client resources are common to all editions except that advanced security features are disabled.

  SonicMQ Developer Edition requires no additional resources and will get you working with SonicMQ quickly. As a result, you will be better prepared to move to the next step: developing your complete application with SonicMQ Professional Developer Edition and then deploying the messaging server with the SonicMQ E-Business Edition.

- **SonicMQ Professional Developer Edition** is a fully-featured development environment. This edition supports unlimited physical connections, the Dynamic Routing Architecture, unlimited clustering, and security with 40-bit encryption and 56-bit encryption but does not support the deployment of applications. When you are ready to deploy, you can

move up to the SonicMQ E-Business Edition. See the *SonicMQ Installation and Administration Guide* for more information about SonicMQ Professional Developer Edition.

- **SonicMQ E-Business Edition** is a fully-featured deployment environment. Designed for complex and high-use deployments, this edition supports unlimited physical connections, the Dynamic Routing Architecture, unlimited clustering, and security with 40-bit encryption and 56-bit encryption. See the *SonicMQ Installation and Administration Guide* for more information about SonicMQ E-Business Edition.

When you realize that SonicMQ is the solution you need for your clients or for your own business, Progress Software Corporation and its partners worldwide are ready to help you plan your development, configuration, and deployment.

First Steps with SonicMQ Developer Edition

Getting started with Progress SonicMQ requires just a few of the common steps for installing applications from media.

Requirements for SonicMQ Developer Edition

SonicMQ Developer Edition has been tested in a well-defined environment to ensure straightforward setup and initialization. As a 100% Java solution, various Java virtual machines and Java versions are compatible with SonicMQ.

Important See the *SonicMQ Release Notes* on the distribution media or **www.sonicmq.com** for more information about the recommended SonicMQ Developer Edition environments:
- Platform specifics.
- Disk and RAM requirements.

The SonicMQ Developer Edition software contains all the required software and documentation for installing, setting up, and running SonicMQ Developer Edition including:

- **Java Runtime Environment** — The preferred JRE is installed and set up for easy access by the SonicMQ Developer Edition.

Other Java Runtime Environments might be packaged on the distribution media. See the *SonicMQ Release Notes* for information about installing and setting up any of these alternate virtual machines.

- **XML Parser** —The IBM® XML Parser for Java Edition is installed and set up for easy access by the SonicMQ Developer Edition.

Additional Software

While the preferred Java Runtime Environment is set during installation, you might want to access other Java resources, as listed in Table 3, to modify the samples and develop new JMS applications.

Important See the *SonicMQ Release Notes* or `www.sonicmq.com` for the latest data about Java resources.

Table 3. Java Resources

Type	*Comments*
Java Compiler	No compiler is required or installed with the SonicMQ distribution software. The samples consider `notepad.exe` or `vi` as the basic editor.
JavaSoft Java Development Kit (JDK)	If you have the appropriate JavaSoft JDK installed on your system, you can use its resources.
Java Virtual Machines (JVM)	Multiple instances of disparate JVMs can run concurrently on a single system.
Java Development Environment (JDE)	Commercially available Java development environments can be used to modify and compile sample files and any Java files you create.

Deciding How to Setup the SonicMQ Developer Edition

The SonicMQ Developer Edition let you explore SonicMQ message server capability fully and completely—excepting advanced security—from a set of client applications on a single computer.

The Developer Edition can use a very robust system in terms of disk and memory to perform effectively under some setups. This is because the only real constraint in the this edition is that it accepts clients from a single IP address. The first client connection to a server accepting connections declares the accepted IP address for that server process. The following figures and discussions explore what set ups are possible in the Developer Edition.

In Figure 17, the basic installation of the Developer Edition installs one message server then creates several client applications that connect to that server, all on the same system.

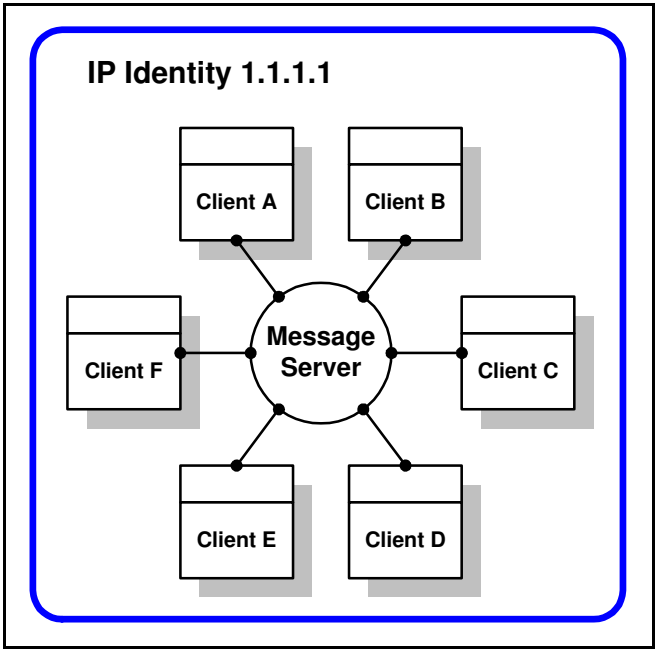


Figure 17. A System with the Developer Edition Server and Clients

You can extend the message server by installing more instances of the message server on the same system and then relating them together to form a cluster. Under that setup, you could create the cluster on one system and connect from another system, as shown in Figure 18.

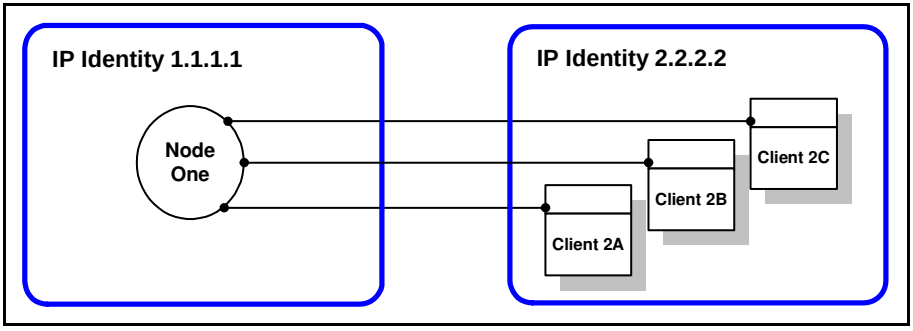


Figure 18. Two Systems With the Developer Edition Node and Clients

If you have the system resources and want to explore multiple nodes, you can create multiple nodes and clients on one system, as shown in Figure 19.

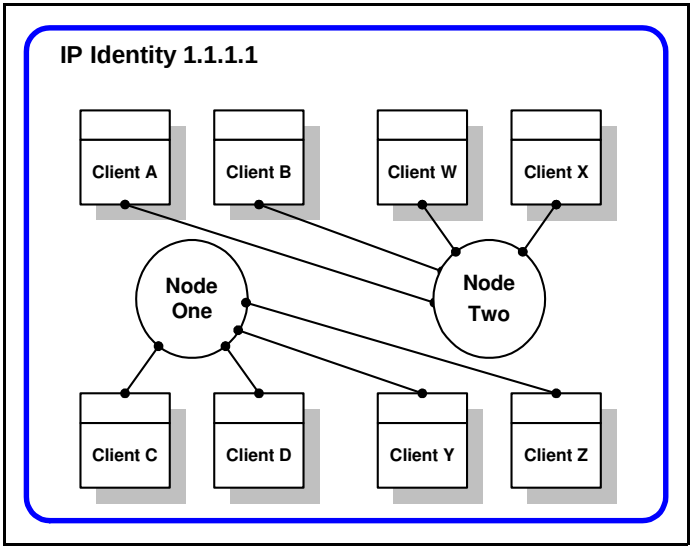


Figure 19. Two Developer Edition Nodes on One System

When you want multiple nodes and want to get beyond a single local host, you can alternate the systems where the nodes and clients reside.

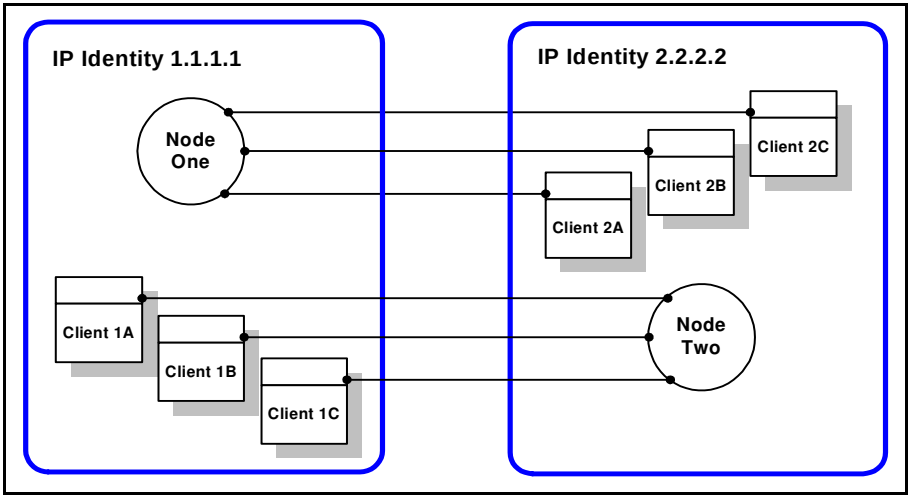


Figure 20. Two SonicMQ Developer Edition Nodes on Two Systems

When you decide how you are going to install the SonicMQ Developer Edition, proceed to the registration and installation procedures.

Installing SonicMQ Developer Edition

If you have the required hardware, you are ready to install the SonicMQ software. This installation of SonicMQ Developer Edition software is pre-configured to use the embedded database that is packaged with the installer. Consult your Progress Software representative or `http://www.sonicmq.com` if you want to know more about other platforms and databases.

➤ To get a license key for a SonicMQ Developer Edition installation

You need a license key to extract the software from the media.

- ☐ Register your SonicMQ Developer Edition at `http://www.sonicmq.com`.

As soon as you register your SonicMQ product and provide your email address, your license key will be sent to your e-mail address.
You will also gain access to the information and services for registered users

including download access for the latest documentation, software upgrades, and the online Developers Exchange.

➤ To install Progress SonicMQ Developer Edition from a CD

1. Insert the SonicMQ distribution CD into the system where you want to perform a SonicMQ installation.
2. Use the Windows Explorer to locate `setup.bat` on the distribution media.
3. Double-click on the `setup` file to launch it. The SonicMQ Installer wizard opens.
4. Follow the wizard prompts for installing SonicMQ.
5. When requested, enter the license key for your installation.

➤ To install Progress SonicMQ Developer Edition from a downloaded file

1. Use the Windows Explorer to locate the downloaded executable on your system.
2. Double-click on the downloaded executable file name to launch it. The SonicMQ Installer wizard opens.
3. Follow the wizard prompts for installing SonicMQ.
4. When requested, enter the license key for your installation. When installation is finished the documentation portal page displays in your default browser.

After the Installation Is Complete

When the Progress SonicMQ Developer Edition installation has completed successfully, the software is ready to run. You do not need to reboot the machine. The documentation page is displayed in your default browser.

The installed SonicMQ Developer Edition has the following characteristics:

- **Connections are limited** — One hundred concurrent client connections allowed, all initiating connection from a single IP address.
- **Security is not enabled** — While you could reinitialize the database to implement security, the basic install defers security to a later exercise after

exploring the samples. See the *SonicMQ Programming Guide* to learn how to extend the samples by implementing security.

- **Message Server database is ready —** The database drivers are installed and the embedded database instance has been initialized.
- **Windows Start menu commands** — The Windows Start Menu commands are shown in Figure 21.

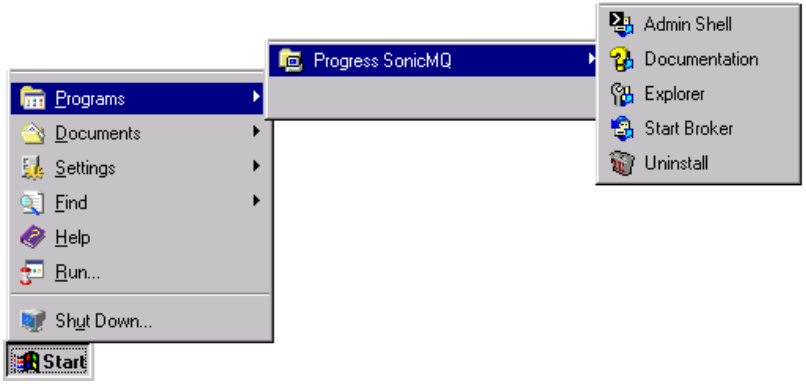


Figure 21. SonicMQ Developer Edition Start Menu Commands

The Windows Start menu commands are:

– **Admin Shell** — Opens the Admin command-line console window where you can enter administrator functions.

– **Documentation** — Accesses the Progress SonicMQ help documentation portal that links to all the online documentation and related Web sites.

– **Explorer** — Opens the administrator Java window for listing and maintaining administered objects, security objects, and configurations.

– **Start Broker** — Starts the SonicMQ Message Server.

– **Uninstall** — Starts the process to remove the Progress SonicMQ installation. You are asked to confirm this action to start the process.

Starting the Message Server

You can now start the SonicMQ Message Server from the Windows Start menu or from a Command Prompt window.

➤ **Starting the message server from the Windows Start menu**

- ☐ Choose **Start > Programs > Progress SonicMQ > Start Broker**.

➤ **Starting the message server from a Windows console window**

- ☐ In a new Command Prompt window set to the SonicMQ install directory, type **`startbr.bat`** and press **Enter**.

➤ **Starting the message server from a UNIX or Linux console window**

- ☐ In a new console window set to the SonicMQ install directory, type **`startbr.sh`** and press **Return**.

The Start Broker console opens and displays its information as the server starts up. The message server is ready when it displays:

SonicMQ Broker started, now accepting tcp connections on port 2506...

You can minimize the Command Prompt window. Closing the window, however, will stop the message server.

What's Next?

When the message server is accepting, you can:

- Run the SonicMQ sample applications
- Take a minute to work with the SonicMQ Explorer

Running the SonicMQ Client Samples

The sample files provided with SonicMQ are ready to demonstrate the JMS concepts on the server that is now running. See Chapter 3, "SonicMQ at Work" to start using these samples.

Using the SonicMQ Explorer

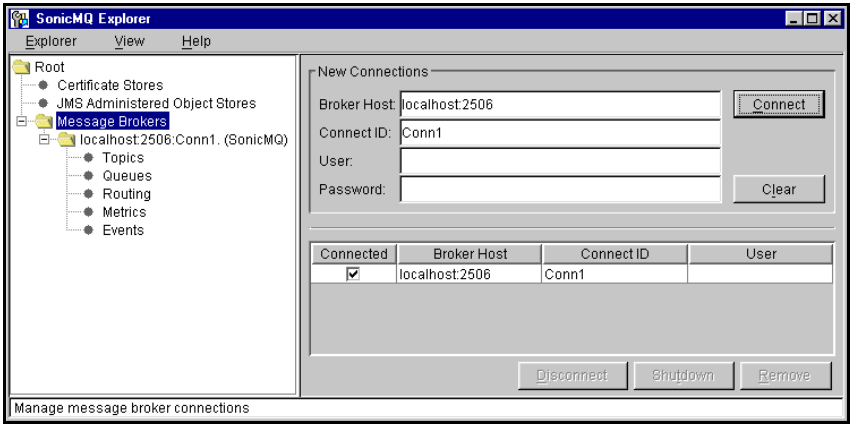
The SonicMQ Explorer is a client that connects to a specific server. Designed to perform administrator tasks as well as client tasks, the Explorer becomes more valuable when you have created some content and better understand some of the JMS client functions. To experience how the SonicMQ Explorer can provide insight into messaging features, do the following procedures.

➤ Procedure 1: Start the SonicMQ Explorer under Windows NT

- ☐ Choose **Start > Programs > Progress SonicMQ > Explorer.**
  The SonicMQ Explorer window opens at its root level.

➤ Procedure 2: Create a Connection

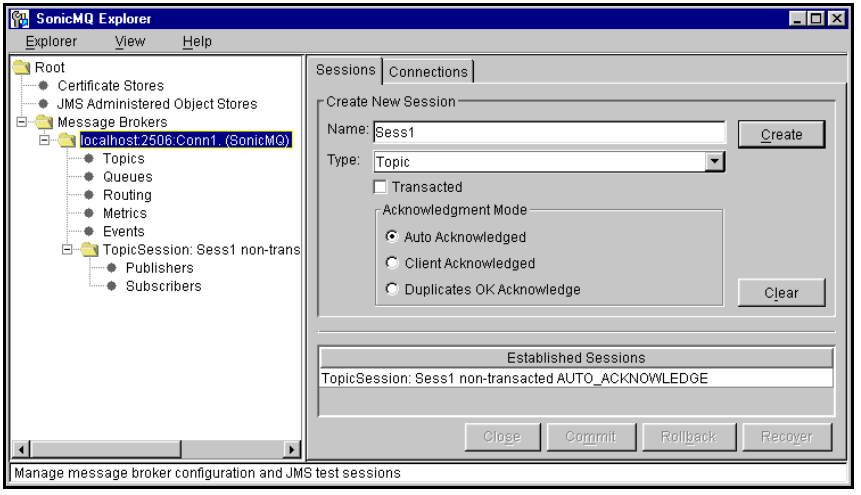
1. Click on **Message Brokers** in the Explorer tree.
2. Enter Broker Host **`localhost:2506`**.
3. Type **`Conn1`** as the ConnectID.
4. Choose **Connect**. The Explorer view shows the connection:



➤ Procedure 3: Create a Session on the Connection

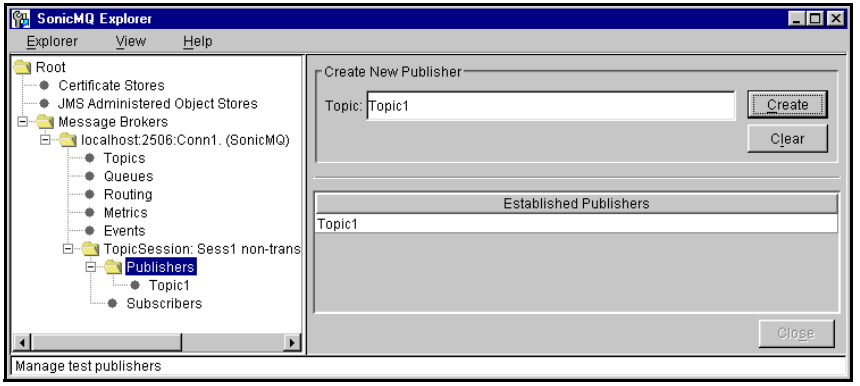
1. Click on the server you just connected to: `localhost:2506:Conn1`.
2. Type **`Sess1`** for the Name of the new session.

3. Click **Create**. The Explorer view shows the new session:



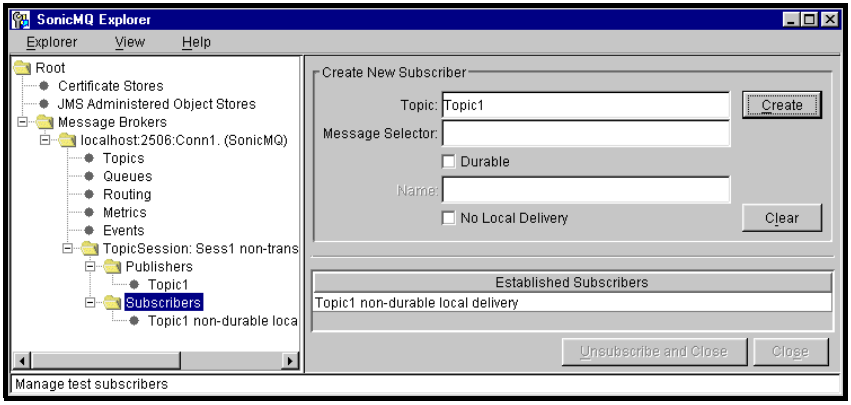
➤ Procedure 4: Create a Publisher in the Session

1. Click on **Publishers.**
2. Type **`Topic1`** as the Topic.
3. Click **Create**. The Explorer view shows the new publisher:



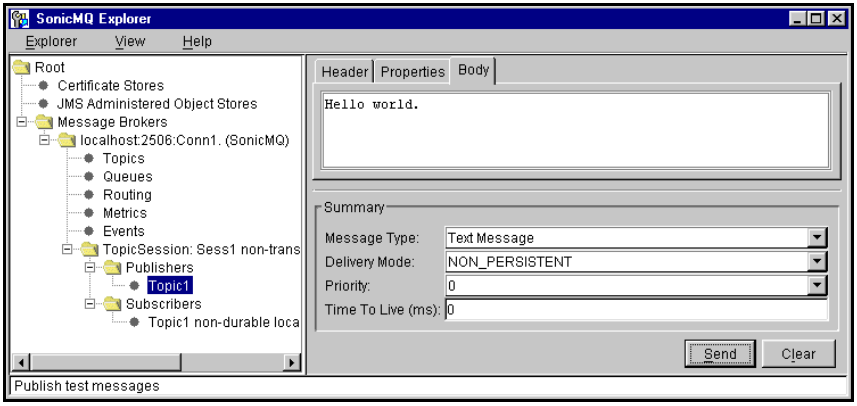
➤ Procedure 5: Create a Subscriber in the Session

1. Click on **Subscribers.**
2. Type **`Topic1`** as the Topic then click **Create**. The Explorer view shows the new subscriber:



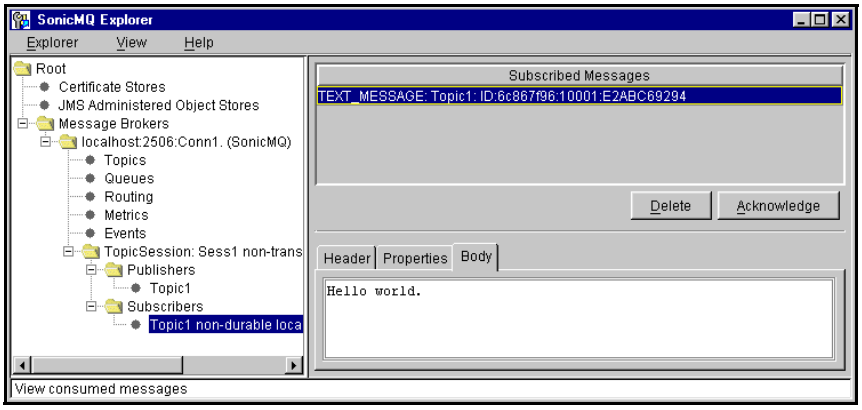
➤ Procedure 6: Publish a simple Text message

1. Click on the line in the tree you just created: **Publisher:Topic1**.
2. Click the **Body** tab, click in the text box and then type **`Hello world.`**
3. Click **Send**. The Explorer view shows the new Text message:



➤ Procedure 7: View the message received as a Subscriber to the Topic

1. Click on the **Subscriber:Topic1** you just created.
2. Select the one Received Message in the list.
3. Click the **Body** tab. `Hello world.` displays.



Closing the SonicMQ Explorer

After completing the Explorer set of procedures, feel free to explore the header, properties, and body of a message. Before you go ahead to the next chapter to see how the sample applications present the features of SonicMQ, close the SonicMQ Explorer to recover concurrent clients and free system resources.

➤ To close the SonicMQ Explorer

The Explorer is a client. Closing the Explorer does not stop the message server.

- ☐ In the SonicMQ Explorer window, click the close box:

Note The basic Explorer functions are available for any Explorer login until the administrator denies these privileges. Many other Explorer functions are reserved for administrators. See the *SonicMQ Installation and Administration Guide* for more information about the SonicMQ Explorer.

Chapter 3

SonicMQ at Work

Trying Out the SonicMQ Samples

After you have successfully installed and started SonicMQ, it is time to get acquainted with what SonicMQ can do for your business. The SonicMQ samples are simple applications that demonstrate typical messaging features.

The samples in this book are grouped into sets of samples for each messaging domain. Publish and Subscribe samples are presented first followed by the Point-to-point samples.

Publish and Subscribe Domains (TopicPubSub Folder)

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

- **Chat Application** — When sessions are running **`Chat`**, a message entered in one session is displayed in all windows.
- **Message Monitor** — The **`MessageMonitor`** sample 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`**, but if one of the sessions is interrupted, messages are retained for it by the message server.
- **Reliable Chat Application** — The **`ReliableChat`** sample application shows how to implement tactics that make connections more resilient.

- **Selector Chat Application** — Message delivery is constrained by the defined message selection criteria in the **`SelectorChat`** sample.
- **Hierarchical Chat Application** — The **`HierarchicalChat`** sample application demonstrates the advantages of using SonicMQ topic trees over message selectors.
- **Transacted Session** — In a **`TransactedChat`** 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 publish it to the topic where the `Requestor` said it would wait for reply.
- **XML Messages** — In **`XMLChat`**, messages are translated into XML format for publication and then interpreted when received by a subscriber.

Queue Point-to-Point Domains (QueuePTP Folder)

In the Point-to-Point domain, replicas of the Pub/Sub samples are run in multiple console windows to see how messages are produced and consumed through queues:

- **Talk Application** — When sessions are running **`Talk`**, a message entered in one session is displayed in one other receiver window.
- **Queue Monitor** — The **`QueueMonitor`** sample uses a Java window to review the messages waiting on a specified queue. While sessions are running `Talk` without queue receivers, the messages that are waiting in the queue are browsed.
- **ReliableTalk Application** — In **`ReliableTalk`**, a series of messages are sent and received under specified settings to reveal basic performance in the Point-to-point messaging domain.
- **Selector Talk Application** — Message delivery is constrained by the defined message selection criteria in the **`SelectorTalk`** sample.
- **Transacted Session** — In **`TransactedTalk`**, a set of entries is stored 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 would wait for reply.
- **XML Messages** — **`XMLTalk`** messages are translated into XML format for publication and then interpreted when received by the message's consumer.
- **Map Messages** — **`MapTalk`** messages are translated into map format—name-value pairs—for publication and then interpreted when received by the message's consumer.
- **Queue Round Trip Application** — In **`RoundTripTalk`**, a series of messages are sent and received under specified settings to reveal basic performance in the Point-to-point messaging domain.

Using the SonicMQ Samples

Important The SonicMQ samples in this book are attuned to the SonicMQ Developer Edition and the standards used on a Windows system.

If you installed the **Professional Developer** or the **E-Business Edition**:

- **On Windows** — The samples in this chapter work as described.
- **With a Security database initialized** — The samples in this chapter work as described if *`usernames`* with a password are set up in the security database.
- **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`**).
  - Substitute a forward slash (**`/`**) for any instance of a backslash (****).

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

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

Starting the Message Server

If the message server is not already running, start the SonicMQ message server before running any of the samples. The following procedures are appropriate for all types of SonicMQ installations.

➤ **Starting the message server process from the Windows Start menu**

- ☐ Choose **Start > Programs > Progress SonicMQ > Start Broker**.

➤ **Starting the message server process from a Linux or UNIX console window**

- ☐ In a new console window set to the SonicMQ install directory, type **`startbr.sh`** and press **Return**.

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

Warning You can minimize the console window. Closing the window, however, will stop the message server.

Client Console Windows

Each sample application instance is intended to run in its own console window with the current path in the selected sample directory, for example, **`<install-dir>\samples\TopicPubSub\Chat`**

You can stop a sample application and reuse the console window.

Using the Sample Scripts

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

```
<install-dir>\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, sets the local **`CLASSPATH`**, and then invokes the executable with its parameters, the **`CLASSPATH`**, and a list of variables. The script runs the sample applications but you might need to adjust it if you use long parameter lists. Standard invocation of the script from a sample folder is 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.

Topic Publish and Subscribe Samples

Chat Application

The `Chat` application shows the most basic Publish and Subscribe activity: whenever anyone sends a text message to a given topic, all active users receive that message as subscribers to that topic.

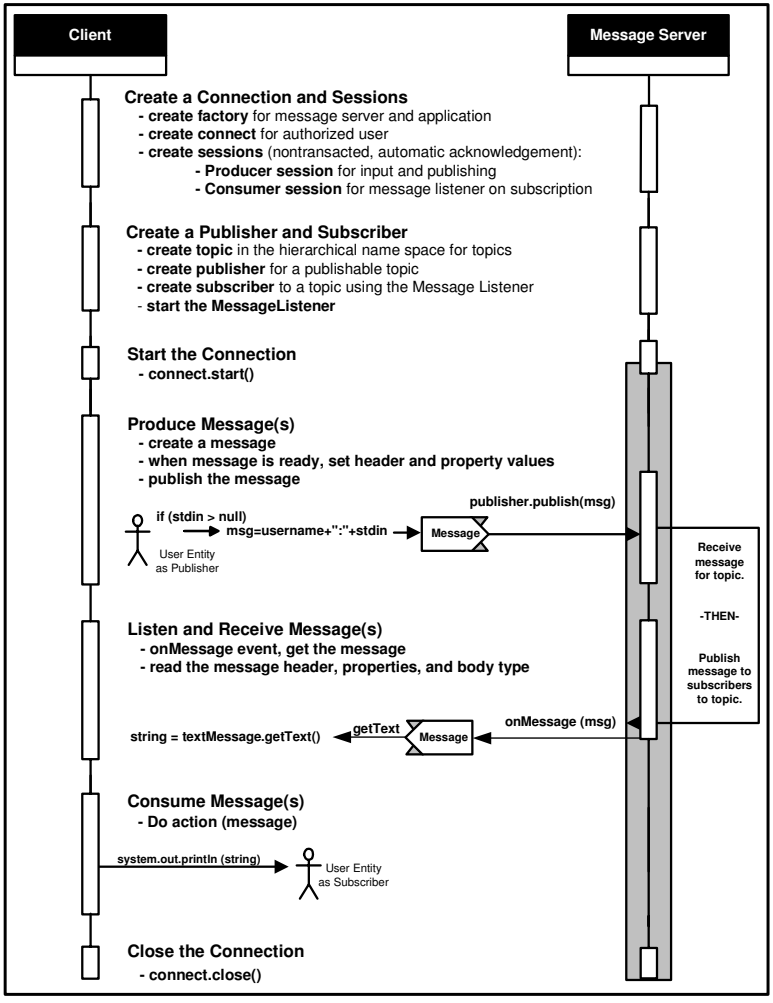


Figure 22. Diagram of the Chat Application Functions

In the `Chat` Session diagram shown in Figure 22, each of the groups of steps describes a process in the `Chat` application running on the client system while it communicates with the message server system. The processes that involve both the client and the message server are noted with process blocks. The extent of the active connection is highlighted in gray.

➤ Starting Chat

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

Note User names cannot contain the reserved characters period (**`.`**), pound (**`#`**), dollar sign (**`$`**), or asterisk (**`*`**).

➤ Chatting

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 arrays. If they miss some of the messages, they just pick up the latest whenever they reconnect to the message server. The message volume could be huge, 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 start a Java window that will monitor the message traffic.

Message Monitor

An example of a supervisory application with a graphical interface is `MessageMonitor` where the application listens for any message topic activity—by subscribing to all topics in the topic hierarchy—and then displays each message in its window:

- **What messages are displayed?** Messages that have been delivered.
- **When does the display update?** When a message is published to a subscribed topic, it is added to the displayed list.
- **What happens when the message server and monitor are restarted?** As messages are listed at the moment they are delivered, there are no messages in the **MessageMonitor** until new deliveries occur.

➤ Running MessageMonitor

1. Open a console window to the `MessageMonitor` folder then enter:
   `..\..\SonicMQ MessageMonitor`

   The console window indicates that it has subscribed to **#**, the wildcard character that means that all topics at all levels will be displayed.

The `MessageMonitor` window opens.

Now send some `Chat` messages.

➤ Chatting

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

The message traffic displays in the window. Figure 23 shows how several messages might appear in the MessageMonitor window that were published to the `Chat` topic, `jms.samples.chat`.

Figure 23. 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 that were sent while they were inactive. They create **durable subscriptions**, permanent records in the message server's database. The `DurableChat` sample publishes its messages to the topic `jms.samples.durablechat`.

Whenever a subscriber reconnects to the topic under the registered name, all undelivered messages to that topic that are still alive will be delivered immediately. The administrator can terminate durable subscriptions or a client can use the `unsubscribe` method to close the durable subscription.

Figure 24 shows what occurs when the subscriber requests an extra effort to ensure delivery.

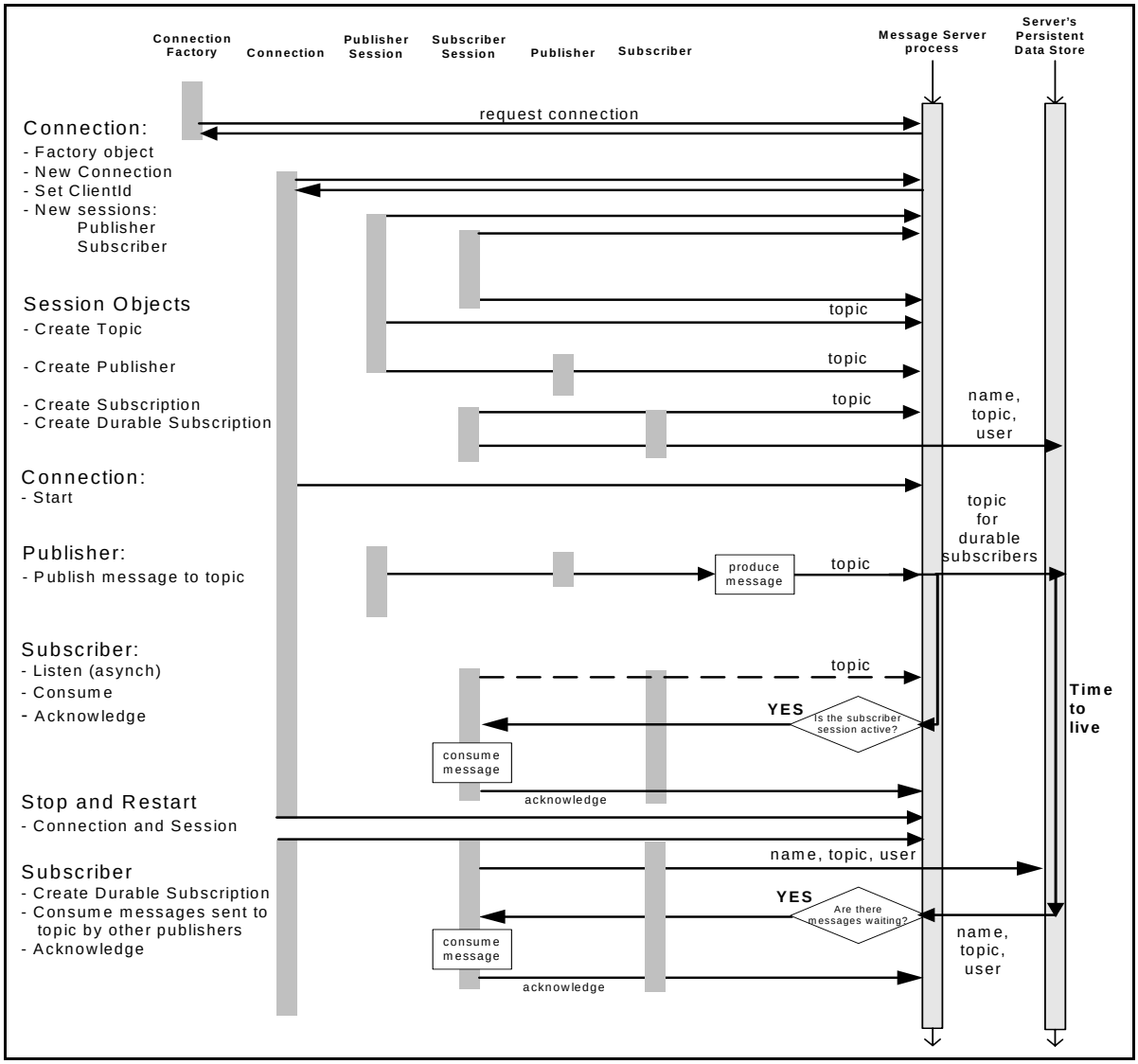


Figure 24. Sequence Diagram for the DurableChat Application

➤ Starting DurableChat Sessions

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

   When the `DurableChat` session reconnects, the retained messages are delivered and then displayed unless the messages have expired. The publisher of the message decided to set the **time-to-live** parameter to 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 in their sessions.

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

Note While at least one hundred clients can be active on any edition of SonicMQ, the necessity of running each console window under a separate JVM instance can exceed memory capacity when five or ten windows are running samples.

➤ Stopping Sessions

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

Reliable Chat Application

`The ReliableChat` sample ensures the robustness of the JMS connection by monitoring the connection for exceptions, and re-establishing the connection if it has been dropped.

➤ Starting a ReliableChat Session

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

   ```
   ..\..\SonicMQ ReliableChat -u MustBeConnected
   ```

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

Note The next step presents an aggressive technique for emulating an unexpected message server interruption. The proper technique for shutdown is to log in the SonicMQ Explorer to the message server and then choose **Shutdown**. If you are demonstrating this sample in a production environment, providing an orderly shutdown will still demonstrate the intended application behavior.

3. Stop the message server by pressing **Ctrl+C** in the message server window. The connection is broken. The **`ReliableChat`** application tries repeatedly to reconnect.
4. Restart the message server by using its Windows **Start** menu command or the **`startbr`** script. The **`ReliableChat`** application reconnects.

You can stop the `ReliableChat` session before proceeding to the next sample.

➤ Stopping ReliableChat

- ☐ In the console window, press **Ctrl+C**. The application stops.

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 (**`-s`**) that it wants to apply.

➤ Starting 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`

➤ Chatting

1. In one of the **`SelectorChat`** windows, type text and then press **Enter**. The text is only displayed in that **`SelectorChat`** window. The subscriber is selecting messages based on a different selector string.
2. Stop the **`Support`** session by pressing **Ctrl+C**.
3. Restart the session, changing the selector string, as follows:
   `..\..\SonicMQ SelectorChat -u Support -s Specs` and press **Enter**.
4. 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` sessions before proceeding to the next sample.

➤ Stopping SelectorChat sessions

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

Hierarchical Chat Application

SonicMQ's lets an application have the power of a message selector 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 publish topic (**-t**) and one for the subscribe topic (**-s**).

➤ Starting HierarchicalChat Sessions

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

➤ Chatting

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.

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

➤ Stopping HierarchicalChat Sessions

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

TransactedChat Sessions

Transacted messages involve a session where groups of messages are buffered on the message server until either of two conditions is met:

- The signal to **commit** the set of messages (in this sample, the string **`OVER)`** is entered. To commit, the series of messages held by the message server is released serially to subscribers, although not as a bound set.
- The signal to **roll back** the set of messages (in this sample, the string **`OOPS!`**) is entered. To roll back, the message server is told to flush the series of held messages without sending them to anyone.

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

Note **Multiple Sessions** — The `TransactedChat` sample wants to be able to commit or roll back published messages but not affect messages received by subscribers. The application creates two sessions—`publishSession` and `subscribeSession`—on the one topic connection.

➤ Starting TransactedChat

1. Open a console window to the `TransactedChat` folder.
2. Enter: **`..\..\SonicMQ TransactedChat -u Audit`**
3. Open another console window to the `TransactedChat` folder.
4. Enter: **`..\..\SonicMQ TransactedChat -u Orders`**

➤ Building a Transaction Then Committing It

1. In one of the `TransactedChat` windows, type text and then press **Enter**. Notice that the text is not displayed in the other `TransactedChat` windows.
2. Again enter text in that window and then press **Enter**.
   The text is still not displayed in the other `TransactedChat` windows.
3. Enter **`OVER`**

All of the lines you had entered are published to a topic and then delivered to subscribers with the transaction setting marked **`true`**. The transaction buffer is cleared. Subsequent entries will accrue into a new transaction.

➤ Building a Transaction Then Rolling It Back

1. In one of the `TransactedChat` windows, type text and then press **Enter**.
2. Again enter text in that window and then press **Enter**.
3. Enter **`OOPS!`**

Nothing is published. The transaction buffer is cleared. Subsequent entries will accrue into a new transaction. After a commit or rollback, the session begins accruing messages again for the next transaction.

All the samples that are running can be stopped now.

➤ Stopping TransactedChat sessions

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

Request and Reply

While the producer of a message can provide for long-lived messages to durable subscribers and acknowledgement from the message server, neither of these techniques confirms to the sender that a message was delivered. To solve this, SonicMQ lets you request a reply when a message is sent. The request sets up a **temporary topic** for that request, and then the header information compels the subscriber to publish a reply to the publisher of the original message. A correlation identifier can be used to coordinate the activities.

In this simple example the replier is set up to simply *fold the case*—receive text and send back the same text as either all uppercase characters or all lowercase characters—of the requestor's message and then publish the message to the temporary topic that was set up for the reply.

Warning Start the replier before the requestor so that the replier's message listener can receive the message and release the blocked requestor.

➤ Starting the Pub/Sub Replier

1. Open a console window to the `RequestReply` folder.
2. Enter: **`..\..\SonicMQ Replier`**

   The default value is allowed for the result mode: **`uppercase`**.

➤ Starting the Pub/Sub Requestor

1. Open another console window to the `RequestReply` folder.
2. Enter: **`..\..\SonicMQ Requestor`**

➤ Testing a Pub/Sub request and reply

- ☐ In the `Requestor` window, enter **`AaBbCc`** and then press **Enter**.

  The `Replier` window reflects the activity, displaying:

  [Request] SampleRequestor: AaBbCc

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

  [Reply] Uppercasing-SAMPLEREQUESTOR: AABBCC

All the samples that are running can be stopped now.

➤ Stopping the Pub/Sub Replier

- ☐ In a `Replier` window, type **`EXIT`**, and press **Enter**. The application cleans up the resources then closes the connection.

➤ Stopping the Pub/Sub Requestor

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

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

XML Messages

XML documents are text documents composed of tagged text blocks that provide a logical way to interpret the content of the message. An XML processor, acting on behalf of an application, interprets the data structure by parsing both the standard XML syntax and the Document Type Definitions (DTD's)—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 XML documents specified by the World Wide Web Consortium. It defines the logical structure of documents and the way a document is accessed and manipulated. The DOM is an object model that represents XML documents in an application-independent form as a hierarchy of objects. The sample application takes the input text, formulates it into XML syntax, and displays it as DOM nodes.

➤ Starting an XMLChat Publisher

- ☐ Open a console window to the `XMLChat` folder then enter:
  `..\..\SonicMQ XMLChat -u Catalog_Update`

➤ Starting an XMLChat Subscriber

- ☐ Open another console window to the `XMLChat` folder then enter:
  `..\..\SonicMQ XMLChat -u Aggregator`

➤ Starting a Chat Subscriber

- ☐ Open a console window to the `Chat` folder then enter:
  `..\..\SonicMQ Chat -u Just_Text`

➤ Sending 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 simple yet complete XML syntax.
  - 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>
```

➤ Sending 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`** in its prescribed way:
    `[TextMessage] Just_Text: Hello`
  - The **`Just_Text`** `Chat` window displays the text in its usual way:
    `Just_Text: Hello`

➤ Stopping a Sample

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

All the samples that are running can be stopped now.

Queue Point-to-Point Samples

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 assume that the sample queues were set up in the message server database when SonicMQ was installed.

➤ To Review the Sample Queues

1. In Windows, choose **Start > Programs > Progress SonicMQ > Explorer.** The SonicMQ Explorer window opens at root level.
2. Click on **Message Brokers** in the Explorer tree.
3. Type **`localhost:2506`** in the Broker Host text box.
4. Enter any Connect ID text such as **`Conn1`** then choose **Connect**.
5. Click on the message server you just connected to: `localhost:2506:Conn1`.
6. Click on `Queues`. The Explorer view lists the sample queues (`SampleQ1`, `SampleQ2`, `SampleQ3`, and `SampleQ4`). Notice that each queue's **Exclusive** option is cleared to allow multiple concurrent receivers on that queue.

Warning If the sample queues are not listed, you need to create the queues. See the *SonicMQ Installation and Administration Guide* for information about setting up queues.

You can close the Explorer before continuing with the samples.

Talk Application

The `Talk` application seems similar to the `Chat` application where the Publish and Subscribe paradigm is used. The difference that is immediately noticeable is that the queue names are parameters when the `Talk` application is started. You must specify either or both a queue for sending and one for receiving.

In `Talk`, two console sessions are started, each with a receiver and a sender where one's sender queue is the other's receiver queue. Then, when you type text and press **Enter**, the message is sent only to the indicated `Talk` partner. If you started several receivers, only one of them would receive the message.

➤ Starting 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`

➤ Talking

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

Stop the samples that are running now.

➤ Stopping a Sample

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

Queue Monitor

The QueueMonitor displays messages much like the Message Monitor sample shown in the Topic Pub/Sub sample, `MessageMonitor`. But the nature of the two monitors underscores fundamental differences between the two messaging models:

- **What messages are displayed?** Messages that are not yet delivered.
- **When does the display update?** When you click the **Browse Queues** button, the list is refreshed.
- **When does the message go away?** When the message is delivered (or when it expires.)
- **What happens when the message server and monitor are restarted?** Listed messages are in the message server database if they were sent with the delivery mode **`PERSISTENT`**. These messages are redisplayed when the message server and the QueueMonitor restart and then you choose to browse queues.

➤ Starting 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.

➤ Starting a Talk Session without a Receiver

Start a `Talk` application sending to **`SampleQ1`** but no receiver queue.

1. Open a console window to the `QueuePTP\Talk` folder.
2. Enter: **`..\..\SonicMQ Talk -u RFP -qs SampleQ1`**

➤ Queuing Messages and Browsing the Queue

1. In the `Talk` 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** Java window, click **Browse Queues** to scan the queues and display their contents.
5. The QueueMonitor list the messages in a display similar to the window shown in Figure 25.

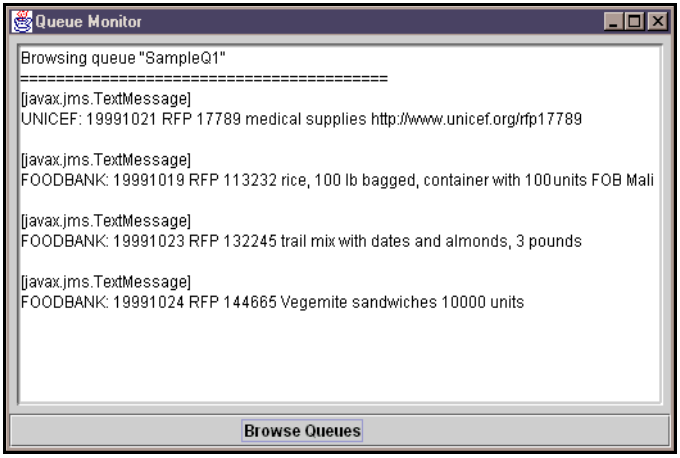


Figure 25. QueueMonitor Window

➤ Receiving the Queued Messages

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

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

1. Open a console window to the `Talk` folder.
2. Enter: **`..\..\SonicMQ Talk -u FlushQ1 -qr SampleQ1`**

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

You can close the QueueMonitor if you need to conserve memory resources.

➤ Stopping the QueueMonitor

☐ Close the **QueueMonitor** window.

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 has been dropped.

➤ Starting 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 message server and then returned to the client as a receiver on that queue. The connection is active.

Note The next step presents an aggressive technique for emulating an unexpected message server interruption. The proper technique for shutdown is to log in the SonicMQ Explorer to the message server and then choose **Shutdown**. If you are demonstrating this sample in a production environment, providing an orderly shutdown will still demonstrate the intended application behavior.

3. Stop the message server by pressing **Ctrl+C** in the message server window. The connection is broken. The **`ReliableTalk`** application tries repeatedly to reconnect.
4. Restart the message server by using its Windows **Start** menu command or the **`startbr`** script. The **`ReliableTalk`** application reconnects.

Stop the `ReliableTalk` session before proceeding to the next sample.

➤ Stopping ReliableTalk

- ☐ In the console window, press **Ctrl+C**. The application stops.

Selector Talk Application

The `SelectorTalk` application demonstrates sending messages using Point-to-point (Queues) with a Message Selector. When messages are sent to a queue, a property is set in the message header to a value specified on the command line. A separate command line value (**`-s`**) is used as a message selector for messages in the receive queue.

➤ Starting SelectorTalk

1. Open a console window to the `QueuePTP\SelectorTalk` folder then enter:
   `..\..\SonicMQ SelectorTalk -u Factory -s Specs -qr SampleQ1 -qs SampleQ2`
2. Open another console window to the `SelectorTalk` folder then enter:
   `..\..\SonicMQ SelectorTalk -u Support -s Recalls -qr SampleQ2 -qs SampleQ1`

➤ Talking

1. In one of the **`SelectorTalk`** windows, type text and then press **Enter**. The text is only displayed in that **`SelectorTalk`** window. The other subscriber is selecting messages based on a different selector string.
2. Stop the **`Support`** session by pressing **Ctrl+C**.
3. Restart the session, changing the selector string by entering:
   `..\..\SonicMQ SelectorTalk -u Support -s Specs -qr SampleQ2 -qs SampleQ1`

   Notice that the messages that were waiting for receivers of specs on **`SampleQ2`** are now delivered to the session.
4. In either **`SelectorTalk`** window, type text and then press **Enter**. The text is displayed in both **`SelectorTalk`** windows. The publisher set the property value to the same value that the subscriber used to select messages.

You can stop the `SelectorTalk` sessions before proceeding to the next sample.

➤ Stopping SelectorTalk sessions

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

Transacted Talk Sessions

Transacted messages involve a session where groups of messages are stored in the queue on the message server until the transactions are either committed as a set or rolled back as a set. The `TransactedTalk` sample intends to commit or roll back messages sent but not affect messages received. The application creates two sessions—`sendSession` and `receiveSession`—on the one queue connection.

When the notice to **commit** is entered, the series of messages held by the message server is released serially to receivers, although not as a bound set. If, at any point, the notice to **roll back** the set of messages is entered, the message server is told 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 may find that the queue receivers get delivery of messages that were not qualified by the selective receivers.

➤ Starting TransactedTalk sessions

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

➤ Building a Transaction Then Committing 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`** windows.
2. Again type text in that window and then press **Enter**.
   The text is still not displayed in the other `TransactedTalk` windows.
3. Enter **`OVER`**

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

➤ Building a Transaction Then Rolling It Back

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

Nothing is published. The transaction buffer is cleared. 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 the `TransactedTalk` sessions before proceeding to the next sample.

➤ Stopping TransactedTalk

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

Request and Reply

In Point-to-point domains, the producer of a message can provide for long-lived messages that persist in the 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 compels the receiver to send a reply to the sender of the original message. A correlation identifier can be used to coordinate the activities.

In this simple example the replier is set up to simply *fold the case*—receive text and send back the same text as either all uppercase characters or all lowercase characters—of the requestor's message and then send the message to the temporary queue that was set up for the reply.

The fundamental difference between the messaging domains is apparent in the expected reply volume:

- **Point-to-point (PTP)**— At most, one reply can occur.
- **Publish and Subscribe (Pub/Sub)** — Any number (none to many) of replies can occur.

➤ Setting up PTP Request Reply sessions

- ☐ Open two (or more) console windows to the `RequestReply` folder.

➤ Starting the PTP Replier

Note It was important in the Pub/Sub `Replier` to prevent blocking by starting the replier before the requestor sends a message. Under PTP, the queue sender (PTP `Requestor`) is not blocked. The requestor can send a message before the replier is available.

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

➤ Starting the PTP Requestor

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

➤ Testing 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 does its operation (converts text to uppercase) and sends the result in a message to the requestor. The requestor window gets the reply from the replier:

  [Reply] Uppercasing-QREQUESTOR: AABBCC

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

➤ Stopping the PTP Replier

- ☐ In a `Replier` window, type **`EXIT`** and then press **Enter**.

➤ Stopping the PTP Requestor

- ☐ In a `Requestor` window, press **Ctrl+C**. The `Requestor` application stops.

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

XML Messages

XML documents are text documents composed of tagged text blocks that provide a logical way to interpret the content of the message. An XML processor, acts on behalf of an application, to interpret the data structure by parsing both the standard XML syntax and the Document Type Definitions (DTD's), rules that define the elements in a document and their relationships.

The Document Object Model (DOM) is an application programming interface (API) for XML documents specified by the World Wide Web Consortium. It defines the logical structure of documents and the way a document is accessed and manipulated. The DOM is an object model that represents XML documents in an application-independent form as a hierarchy of objects. The sample application takes the input text, formulates it into XML syntax, and displays it as DOM nodes.

➤ Starting an XMLTalk Publisher

- ☐ Open a console window to the `XMLTalk` folder then enter:
  `..\..\SonicMQ XMLTalk -u Supplier -qr SampleQ1 -qs SampleQ2`

➤ Starting an XMLTalk Subscriber

- ☐ Open another console window to the `XMLTalk` folder then enter:
  `..\..\SonicMQ XMLTalk -u Picklist -qr SampleQ2 -qs SampleQ1`

➤ Sending an XML Message

- ☐ In the `Supplier` window, enter text—for example,
  `2mm Widget, lot 50000, US$ 300 /M` —and then press **Enter**. The text entry is formulated into simple yet complete XML syntax.

  The **`Aggregator`** window displays the input as nodes in a Document Object Model Element tree:

```
[XML from 'Supplier'] 2mm Widget, lot 50000, US$ 300 /M
ELEMENT: message
 |--NEWLINE
 +--ELEMENT: sender
     |--TEXT_NODE: Supplier
 |--NEWLINE
 +--ELEMENT: content
     |--TEXT_NODE: 2mm Widget, lot 50000, US$ 300 /M
</message>
```

You can stop the `XMLTalk` sessions before proceeding to the next sample.

➤ Stopping the Samples

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

Map Messages

Map messages are message types that transfer a collection of assigned names and their respective values. The names and values are assigned by `set` methods for the Java primitive data type of the value.

For example:

```
MapMessage.setInt("FiscalYearEnd", 10)
MapMessage.setString("Distribution", "global")
MapMessage.setBoolean("LineOfCredit", TRUE)
```

The `MapMessage` name-value pairs are sent in the message body.

Application developers, defining the receiver functions for Map messages, can extract the data from the received message in any order.

The `MapMessage` sample demonstrates that an instance of a `MapMessage` must be received as a `MapMessage` to use its contents. The sample, however, is constrained to accepting standard text input from the console.

Other data types and multiple data elements are easier to demonstrate when you modify the Java source program to perform strongly-typed setting and getting of name-value pairs in a Map message.

See the *SonicMQ Programming Guide* for information about Map messages.

Queue Round Trip Test Application

The `QueueRoundTrip` application sends and receives using multiple sessions and a temporary queue. A temporary queue can guarantee that each instance of this sample receives its own messages only. This sample shows the round trip time for a message being sent to a queue on the message server and then received from that queue.

➤ Starting QueueRoundTrip

When you simply start the text, the default number of messages is **100**.

1. Open a console window to the `QueueRoundTrip` folder then enter:
   `..\..\SonicMQ QueueRoundTrip`
2. Observe the statistics displayed:

```
Sending Messages to Temporary Queue...
Time for 100 sends and receives:                ...ms
Average Time per message:                       ...ms
```

3. Press **Enter.**
4. You can specify the number of messages you prefer, for example **1000**. Enter: **`..\..\SonicMQ QueueRoundTrip -n 1000`**
5. Observe the statistics displayed:

```
Sending Messages to Temporary Queue...
Time for 1000 sends and receives:               ...ms
Average Time per message:                       ...ms
```

6. Press **Enter**.

Stopping Client Sessions and the Message Server

➤ Stopping client sessions and connections

1. In an active client console window, press **Ctrl+C**.

➤ Stopping the message server with the Explorer

1. Choose **Start > Programs > Progress SonicMQ > Explorer**. The SonicMQ Explorer window opens.
2. Click on **Message Brokers** in the Explorer tree.
3. Connect to the Broker Host that you want to shut down, for example **`localhost:2506`**.

The Explorer window appears similar to the one shown in Figure 26.

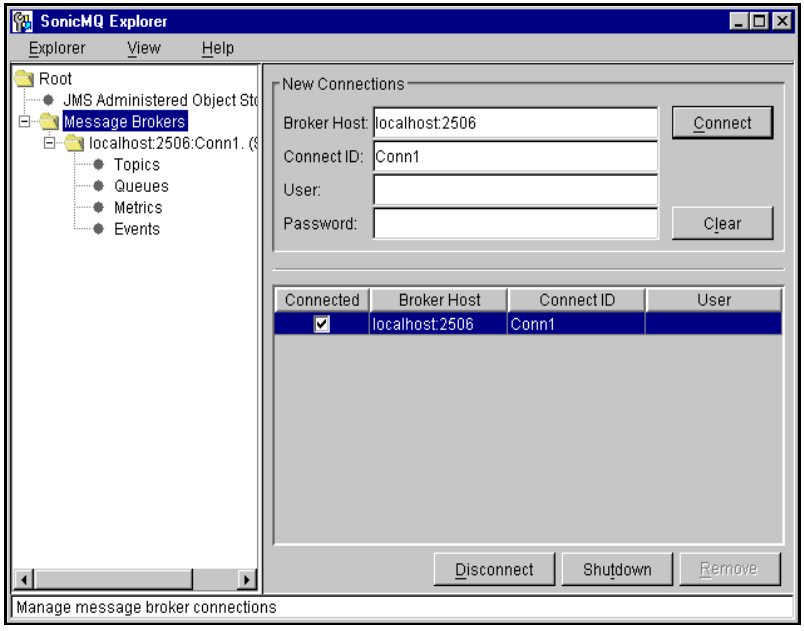


Figure 26. Using the Explorer to Shutdown the Message Server

4. Choose **Shutdown**. The selected SonicMQ Broker Host shuts down.

Chapter 4 Next Steps

Overview

When you complete working through this *Getting Started with SonicMQ* guide, there are several resources where you can learn more about Progress SonicMQ and how Progress Software is committed to delivering performance—not promises—with its e-Business Messaging Server.

Online Books and API Documentation

Open the **Progress SonicMQ Documentation** portal page, **`SonicMQ_Help.htm`**, at the top of the installed SonicMQ directory to access all the distributed documents— everything from the *SonicMQ Release Notes* to PDF-formatted books to the online API files.

Web Sites

The latest SonicMQ information and downloads are always accessible by registering at **http://www.sonicmq.com**. You can also search the KnowledgeBase and visit the SonicMQ Developer Exchange.

Technical and Pre-sales Support

Our professional sales and support staff are ready to help you complete your testing, purchase, and implementation of Progress SonicMQ.

In the USA, call 800.477.6473 ext. 4900 or email: `sonicmqpresales@progress.com` for more information.

Uninstalling SonicMQ

If you need to remove Progress SonicMQ and want to recapture the disk space and resources that the installation uses, follow these steps:

1. Stop all SonicMQ servers and clients.
2. Choose **Start > Programs > Progress SonicMQ > Uninstall**.

   The SonicMQ Uninstall wizard opens.
3. You are asked to confirm that you want to uninstall SonicMQ:
   - If you choose not to uninstall, nothing happens and the wizard closes.
   - If you do want to uninstall, the wizard continues.
4. Follow the uninstaller wizard steps.

When you have completed the uninstall process, Progress SonicMQ is removed from your system.

Glossary

A

Acceptor An object that listens for clients or for other servers.

Acknowledgement Mode In a nontransacted session, the type of acknowledgement a client application expects when a message is delivered. In SonicMQ, the client application can set one of the following for types of acknowledgement:

- `AUTO_ACKNOWLEDGE`
- `CLIENT_ACKNOWLEDGE`
- `DUPS_OK_ACKNOWLEDGE`
- `SINGLE_MESSAGE_ACKNOWLEDGE`

ACL Access Control List. The set of user/group privileges stored for security-enabled SonicMQ servers or clusters.

ActiveX/COM A type of control that allows Windows users to run SonicMQ-enabled components in applications like Progress 4GL, Microsoft Office, Internet Explorer, Lotus® Notes, Lotus SmartSuite®, and Sybase® PowerBuilder.

Adjacent Routing Node Adjacent routing nodes are those nodes that have a server-to-server connection with the current node.

Admin Tool A tool which provides a command-line or script-driven interface for the administration of SonicMQ.

Administered Object Also called JMS administered object. An Object that is defined independently of a SonicMQ server. Within the JMS specification, the objects that can be administered are:

- ConnectionFactories: TopicConnectionFactory, QueueConnectionFactory
- Destinations: Topic, Queue

Administered Object Store A data store for administered objects. The object store can exist in either a file system or a Java Naming and Directory Interface (JNDI) name space.

Administrative Client Connection A special kind of client connection used for Explorer, Admin Tool, or clients that use the Management API.

Administrative Notification A management event the SonicMQ server generates when it finds messages that have expired or that cannot be routed due to a network error.

Administrator A special user created by the installation of SonicMQ. Administrator is established with all permissions on existing topics and queues.

Administrators A group created in the installation of SonicMQ. The Administrators group is established with all permissions on existing topics, and contains the principal user `Administrator`. Only members of this group can use the **Admin>** or Explorer tools to maintain users, groups, and topics in security enabled databases.

Advertising A flag in the routing information of a routing node. When advertising is turned on, information about all known global queues in the node is sent to the routing destination node for dynamic routing configuration purposes. When advertising is turned off, the information is not sent.

Asynchronous Delivery See *Delivery Method*.

Authentication The process by which an entity proves its identity such as providing a password. See *Authorization*.

Authorization The granting of specific rights to a user. Authorization generally involves the administration of an access control list. For authorization to be secure, you must use an authentication mechanism to prevent an attacker from assuming a trusted user's identity. See *Authentication, Access Control List.*

Auto Acknowledge See *Acknowledgement Mode*.

B

B2B Business to Business. (Also B to B.) Loosely-coupled applications in diverse organizations that perform secure business transactions across disparate hardware and software architectures.

Base64 Encoding A method for encoding binary files so they may be transferred easily. For example, base64-encoded binary files can be sent in the body of an email message.

Bastion Host A host machine that is a known entity on the Internet. It is highly exposed and will be the point at which potential attackers will gain access to your internal network and applications. It must be highly fortified.

Binary File See *DER Encoded File*.

Broker The SonicMQ message broker is also referred to as "message server." See *Message Server.*

Bytes Message A stream of uninterpreted bytes. This is useful when target applications cannot read Java types or 16-bit Unicode encodings.

C

Client An application connected through an open connection to a JMS provider's service daemon. Clients may produce messages to, or receive messages from, a destination on a connected message server.

Client Acknowledge See *Acknowledgement Mode*.

Client Connection A connection between a client and a message server or cluster.

Client Identifier (ClientID) An identifier associated with a client that, together with a subscription name that the client assigns and the username, identifies a durable subscription. The ClientID acts as a unique identifier when many clients might be using the same user name and subscription name.

Cluster A collection of inter-connected servers. Each server within the cluster communicates directly with every other server in the cluster.

Commit In a transacted session, sends a series of messages to consumers and disposes of the series of messages consumed since the last call.

Configuration Connection A connection between a message server and its configuration server.

Configuration Server A SonicMQ server that controls the configuration of a server cluster. If security is enabled, the configuration server administers security for the entire cluster.

Connect ID A SonicMQ value that controls whether the server allows multiple connections for users in a client application.

Connect URL An address that a server specifies for incoming routing connections. This is advertised to the remote server when the current server first connects.

Connection A communications thread between a client and a server. Each connection is used either in Pub/Sub messaging or in PTP messaging. The connection is a single point for all communications between the client application and the server.

Connection Factory An administered object that encapsulates a set of configuration parameters. A JMS administrator can create Connection Factories for use by JMS clients. A client application can choose to control the Connection Factories explicitly. In SonicMQ there are two types of Connection Factories:

- In the Publish and Subscribe domain, a Connection Factory is a TopicConnectionFactory. See *Topic, Pub/Sub.*
- In the Point-to-point domain, a Connection Factory is a QueueConnectionFactory. See *Queue, Point-to-Point*.

Connection Protocol The protocol used to establish a connection; either TCP, SSL, or HTTP.

Consumer Recipient of messages sent by a producer to a server destination. The consumer binds to a server destination to receive a message, then implements the message's delivery method. See *Delivery Method, Destination, Producer.*

Correlation Identifier Used in request-and-reply messaging; a designated identifier that certifies that each replier is referred to its original requestor.

D

Dead Message Queue (DMQ) A queue to store messages that are either destined to expire or considered undeliverable. These messages include in-doubt messages and unroutable messages.

Decryption The process of restoring the original text of a message from the encrypted version. See *Encryption*.

Delivery Method The method selected by the client for message delivery. In SonicMQ, clients can select from two types of delivery methods:

- Synchronous delivery — The client requests the next message using a receive method that polls the session's MessageConsumer for a destination, then waits for a reply. A possible disadvantage to this method is that the connection could be blocked indefinitely while the MessageConsumer waits for a reply.
- Asynchronous delivery — The client registers a MessageListener. As messages arrive, the provider calls the listener's `onMessage` method. In the case of an impaired connection, guaranteed messages remain in the message server's database until a connection is re-established. To avoid backlogs, the producer determines how long a message will wait for a consumer.

Delivery Mode A message producer parameter that specifies to the server whether the message is non-persistent (volatile) or persistent. A message's deliveryMode is effective throughout its lifespan. The delivery mode for a message can be one of the following:

- **`NON_PERSISTENT`**
- **`PERSISTENT`**
- **`NON_PERSISTENT_ASYNC`**

De-Militarized Zone (DMZ) A network layer added between the outside network (least secure) and internal network (most secure) in order to add a level of security protection.

DER Encoded File A type of file for holding SSL certificates and private keys. Also referred to as raw, DER encoded, or binary files.

DES Data Encryption Standard. A NIST-standard secret key cryptography method that uses a 56-bit key. DES is based on an IBM algorithm which was further developed by the U. S. National Security Agency. DES is very fast and widely used. See *Triple DES*.

Destination The delivery labels in messaging. The producer sends a message to a destination; the consumer then receives the message from this location. In SonicMQ there are two types of destinations:

- Topic — Producers deliver messages to topics in Pub/Sub messaging. See *Topic*, *Pub/Sub.*
- Queue — Producers deliver messages to queues in Point-to-Point messaging. See *Queue*, *Point-to-Point*.

Digital Certificate An electronic identification that establishes a user's credentials when transacting business on the Web. A certification authority issues a digital certificate. It contains the user's name, a serial number, expiration dates, a copy of the certificate holder's public key (used for encrypting and decrypting messages and digital signatures), and the digital signature of the certificate-issuing authority so that a recipient can verify that the certificate is real.

Direct Access Queue In a trading partner portal application, a server in a portal cluster supports a uniquely named direct access queue that enables applications to address messages back to this server.

DOM Document Object Model. The parsed interpretation of well-formed XML data into a tree structure that is easily incorporated into applications.

Domain The distinctive JMS models of messaging that describe each of the two dominant approaches to messaging currently in use, Point-to-point (PTP) and Publish-and-Subscribe (Pub/Sub).

DTD Document Type Definition. A structured text file that contains information to validate the defined elements and attributes in an XML file.

DTP Distributed Transaction Processing. Covers the reference documents within the specifications covering Communications Resource Manager interfaces for transaction management: TX, CPI-C, XATMI, TxRPC, and XA.

Duplicates OK Acknowledgement Also called Dups_OK. See *Acknowledgement Modes*.

Durable Subscription A subscription where the client wants to receive all messages published on a topic even if the client connection is not active. The server notes the durable subscription and ensures that all messages from the topic's publishers are retained until either they are acknowledged by the durable subscriber or the messages have expired. A durable subscription is not allowed for a temporary topic. See *Inactive Durable Subscription, Subscriber.*

Dynamic Routing A remote server connecting with a routing node may use connections initiated from the routing destination node.

Dynamic Routing Architecture (DRA) A multi-node architecture. Each cluster of servers or unclustered server is a node. DRA provides a robust secure way to send messages from a server on one node to a destination on another node.

E

Encryption Any method of encoding information so that it cannot be read except by the intended recipient. The intended recipient must be able to decrypt the received message.

Enterprise-Level Connection Security A condition where connections are made secure at the enterprise level, not at the level of the ultimate user. This is essential for inter-enterprise secure communication, because one enterprise generally will have no way to authorize and authenticate individual users in the other enterprise.

Events List A table in the SonicMQ Explorer which displays the following types of server events:

- Connect, which records every time a connection is opened
- Disconnect, which records every time a connection is closed
- Drop, which records every time a connection is lost without the client being disconnected, for example, if the client dies
- Reject, which records every time a connection is rejected

Explorer A Graphical User Interface for the administration of SonicMQ.

Exterior Router A router that protects the inside network from the Internet.

F

Failover Connect-time failover is based on a client (or server acting as a client) specifying a list of servers in a cluster to which it may initially connect. If one connection fails, other connections from the list will be tried until either a connection is made, or a timeout condition terminates the attempts.

Firewall A single component or a system of many components that protect a network from other networks, or more specifically protects a private network from the Internet. A private network is a network that you want to protect from attacks. A firewall restricts access to a private network to specific access points.

Flow Control The ability of a message server to refrain from accepting messages sent to a destination when the physical resources allocated to the destination are below a specified level. This technique tries to moderate spikes in message traffic. Applications can choose to either accept the delays and proceed as slowly as traffic permits or throw an exception and handle the result.

G

Global Destination When two routing nodes connect, they can optionally advertise to each other the names of the remote queues that each supports. These become the list of Global Destinations known to a server.

Group A collection of one or more users. See *principal*.

H

Header See *Message Header.*

Hierarchical Namespace A naming technique that SonicMQ uses to allow subscriptions to multiple topics. In a hierarchical namespace, topics are arranged in a tree structure, so each topic (except the root) is the child of another. In SonicMQ, a child topic inherits security from its parent topic. Hierarchical namespaces let the SonicMQ Security Administrator assign security using wildcard matching to specify all children, or all descendents, of a given topic. Hierarchical namespaces can also be used to assign security to queues.

Hop Count The number of routing connections traversed by a message. Each forwarded destination message increments the hop count. This allows shortest paths to be detected and saved for routing.

HTTP Tunneling For security reasons, many firewalls are configured to restrict traffic to a single port, typically port 80, and only if the traffic protocol is HTTP. With HTTP tunneling, traffic is converted to HTTP and directed to port 80 so it can pass through the firewall.

I

Identifier A value returned to indicate a successful or failed function call. A non-negative identifier indicates a successful function call; an identifier with a negative value indicates a failure. Identifiers are defined for the SonicMQ ActiveX/COM control so that its API flattens the object-oriented structure of the JMS API, thus creating a single API.

Inactive Durable Subscription A durable subscription that exists but does not currently have a message consumer subscribed to it. See *Durable Subscription.*

In-doubt Message A message whose delivery has been interrupted by network or hardware failure. In either case, a message has been forwarded to another routing connection, but the handshaking needed to ensure exactly-once delivery does not occur.

Inherit Describes the way that the default security policy of a topic or queue is propagated from the policy of its parents in the topic or queue tree.

Also refers to a particular kind of permission that may be given to a user for publishing, subscribing, sending, or receiving. If a user has Inherit permission, then:

- The user is granted permission if at least one group to which the user belongs has Grant permission.
- The user is denied permission otherwise.

Integrity See *Quality of Protection.*

Interior Router A router that is placed at the point of entry to an inside network.

Interserver Connection A connection between a message server and its peers in a SonicMQ cluster. Servers connected through an interserver connection form a SonicMQ cluster.

J

JMS Java Message Service.

JMS Administered Object An object containing JMS configuration information that is created by a JMS administrator and later used by a JMS client. Defined in the *Java Messaging Service, Version 1.0.2*.

JNDI Java Naming and Directory Interface.

L

Load Balancing A method of distributing connections over several message servers in a cluster to avoid creating a bottleneck that might result from overloading a server.

Local Management Describes the situation in an inter-enterprise messaging system where each enterprise controls access using a locally-maintained list of access control rights.

Log May refer to a server log or a server recovery log. A server log is an optional text file containing output that would otherwise be sent to the console. A server recovery log is a binary file that allows a server to recover its state in the event of a system crash.

Logical Queue An individual physical queue for which you want to specify nondefault security using the Queue Security tab in the Explorer.

Lookup Name A symbolic name which lets a JMS client retrieve the data necessary to make connections to a server and produce and consume. The SonicMQ Admin Tool and SonicMQ Explorer can set up the association between a lookup name and the data.

M

Map Message A set of name-value pairs where names are strings and values are Java primitive types.

Message A package of bytes that encapsulates the message body and then exposes metadata that identify, at a minimum, the following message components:

- timestamp
- priority
- destination
- message type
- message body (for most message types)

Message Authentication Code (MAC) A code used to check the integrity of encrypted file, based on a secret key.

Message Body A set of bytes interpreted as the message type. SonicMQ provides the five message types defined by JMS, and extends the `Text` type to implement the `XML` message type. See *Message Type*.

Message Header Contains values used by both clients and servers to identify and route messages. JMS-standard header fields are JMSCorrelationID, JMSDestination, JMSDeliveryMode, JMSMessageID, JMSTimestamp, JMSReplyTo, JMSRedelivered, JMS Type, JMSExpiration, JMSPriority.

Message Listener Invoked by a topic subscriber to initiate asynchronous monitoring of the session thread for subscriber messages. When the listener is assigned just after the topic subscriber is created for the session, the listener is bound to that topic.

Message Properties Can be any of the following data types: boolean, byte, short, int, long, float, double, String. Customer-defined properties provide name-value pairs that message producers can name, type, populate, and send. Receivers can coerce these custom-defined properties into other acceptable data types. Some property names are reserved for JMS optional properties used by SonicMQ (JMS_SonicMQ_*) such as those used by dead message queues.

Message Receiver A synchronous call to fetch messages for a topic subscriber. In SonicMQ there are three methods to manage message receipt:

- Receive — Receives the nest message produced for the subscriber. This call blocks indefinitely until a message is produced. When a receive method is called in a transacted session, the message remains with the subscriber until the transaction commits.
- Receive with Timeout — Receives the next message within a specified time interval and causes a timeout when the interval has elapsed.
- Receive No Wait — Receives the next available message immediately or instantly times out.

Message Selector Used by message consumers to filter messages. Message selectors evaluate message headers and properties with expression strings created with a subset of SQL-92 semantics. Message selectors do not evaluate message content.

Message Server Listens on a port on its host system to provide server services to its clients. The SonicMQ message server does the following:

- Manages the persistent data store. See *Persistent Data Store*
- Provides simultaneous multi-protocol support for connections (TCP to HTTP & SSL)
- Logs activities

Message Type In SonicMQ there are seven 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 the values are Java primitives
- BytesMessage — Stream of uninterpreted bytes

Metric Generic term for a statistic captured by the Explorer for monitoring a SonicMQ server. SonicMQ captures the following metrics:

- **Memory Usage**
- **Physical Connections**
- **Msgs Rcvd**
- **Msgs Rcvd/sec**
- **Bytes Rcvd/sec**
- **Msgs Dlvd**
- **Msgs Dlvd/sec**

N

Node A single server or cluster of servers where configuration can be centrally administered, and where clients and servers can be seamlessly connected. In the Dynamic Routing Architecture approach, each cluster of servers or unclustered server functions as a node in a routing network.

NT Service A Windows NT feature. Once you have set up a server as a Windows NT service, it can be automatically or manually launched and shut down by using the Services application in the Control Panel group.

O

Object Message A message that contains a serializable Java object.

P

Packet Data to be transferred across a network is broken up into parts called packets. These packets are sent across the wire separately. This system of packets is necessary for many systems to share a network and be fair about sending data between them.

Packet Filtering Router A special kind of router. In addition to asking the question, "Where do I send this packet?" it asks, "Should I send this packet?" It answers this question by checking the security rules that are defined for it. See *Router.*

PBETool A command line tool to DES-encrypt and DES-decrypt a `broker.ini` file.

Permission In SonicMQ the administrator of a security-enabled server may grant or deny various permissions to users or groups. These are permissions to publish or subscribe to a topic and to send or receive to a queue. If a permission is neither granted nor denied, it will be inherited. See *Inherit*.

Persistent Data Store A data source bound to a message server that provides the repository for all the administered data, the security database, and the message store.

Persistent Data SonicMQ has three types of persistent data:

- Message Store — Stores and retrieves queues, persistent queued messages, and messages held for durable subscribers and their subscriptions.
- Security Database — When security is enabled, manages the Access Control Lists for authentication of users and permission for users to read and write to queues and topics.
- Configuration Server — When multi-server activities are enabled, maintains the configuration so that multiple servers can interact, enabling robust and secure server cluster topologies.

Ping A method that lets the application detect when a connection gets dropped by setting an interval to check the presence and alertness of the message server on a connection. This technique is particularly useful for connections that listen for messages, but do not send messages.

Point-to-Point (PTP) A type of messaging, in which a producer delivers a message to a specified static queue at the message server, placing new messages at the back of the queue. Consumers can either receive the first message in the queue 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 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.

Portal A server that has a routing queue. The term Marketplace Portal may be used to describe a portal in some architectures.

Prefetch Count The number of messages that the receiver will take off the queue to buffer locally for consumption and acknowledgment. The default PrefetchCount value is 3.

Prefetch Threshold The minimum number of messages in the local buffer that will allow a new receiver to append more messages to the buffer. The default PrefetchThreshold value is 1.

Principal The term used in SonicMQ for an individual user or group.

Prior Connections Table While connecting to servers, Explorer maintains a list of known hosts, Application IDs, and users in the `explorer.ini` file of the working directory. This data is known as the prior connections table and is read when Explorer starts to initialize the connection table.

Priority A message producer parameter that determines the position of the message in the delivery waiting list. When messages for a message consumer are awaiting delivery, the higher priority messages move to the front of the delivery waiting list, which is otherwise ordered First In First Out (FIFO). See *Queue*.

Privacy See *Quality of Protection*.

Producer Sends messages to a specified destination in the server. These messages are received by consumers from the server destination. The producer packages and encrypts the message body, and identifies the service level and protection for the outbound message. See *Consumer, Destination*.

Proxy Server Software that is in charge of dealing with external (least secure) servers on behalf of internal (most secure) clients. Proxy servers relay requests for information from your site to other sites and then relay requested information back to your site. In short, a proxy server talks with other Internet sites on behalf of your site. This software is often part of a firewall.

PTP See *Point-to-Point*.

Pub/Sub See *Publish and Subscribe*.

PUBLIC A special group that is established when SonicMQ is installed. PUBLIC represents all authenticated users in the system and by default, members in this group are given all permissions on all topics.

Publish and Subscribe (Pub/Sub) In Pub/Sub messaging a producer (publisher) delivers a message to a specified topic at the message server. Consumers (subscribers) subscribe to a topic to receive messages published to that topic. Pub/Sub 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. The publisher declares the quality of service (delivery mode, time-to-live, and priority), as well as whether a reply is requested from a consumer (subscriber).

Q

Quality of Protection (QoP) The type of security protection afforded to a message. SonicMQ allows three levels of QoP:

- None.
- Integrity — The message is guaranteed not to have been changed in transit.
- Privacy — The message is encrypted so that even if it is intercepted, it can not be read as sent. In SonicMQ, whenever a message has privacy it also has integrity.

Quality of Service (QoS) The set of services that exchange time and resource overhead for a deeper commitment to secure, accurate message delivery. The advantage of a broad QoS is that simple information updates can be expedited and crucial business information can be micro-managed.

Queue In PTP messaging, messages are produced to a named queue. New messages are placed at the back of the queue unless they have a specified priority higher than other messages in the queue. Prospective consumers either receive the frontmost message from the queue (thereby removing it from the queue), or browse through all the messages in the queue, causing no changes.

- Global queue — A queue that can be accessed from remote routing nodes.
- Local queue — A queue that can only be reached from direct clients of the server.

See *Point to Point*, *Priority.*

Queue Browser A mechanism that examines queues and allows users to scan messages without destroying them. Users must be authorized to use this tool.

Queue Connection An active PTP connection to a SonicMQ message server.

Queue Group Queue groups are created automatically when servers within a routing node advertise global support for identically named queues, called replicated global queues.

QueueConnection Factory An administered object used to create a connection that binds a PTP client to a message server at runtime.

R

Random List Access A method for accessing a connection URL list. The server to be tried first will be selected randomly. Random start can be used to increase throughput for high-traffic scenarios by not overloading the servers at the start of the list.

Raw File See *DER Encoded File*.

Receiver A consumer in Point-to-point messaging. The message server forwards messages to the receiver from queues where the session is waiting to receive messages. There is one and only one receiver for each message in a queue.

Recover A session's recover method is used to stop a session and restart it with its first unacknowledged message. This method is helpful in the case where a large number of unacknowledged messages has accumulated. Recover methods are most often needed when the acknowledgement mode is `CLIENT_ACKNOWLEDGE`.

Remote Queue A queue used for sending messages to remote routing nodes. A Remote Queue name is composed of a routing node name and a queue name, separated by double-colons (::). You can create a QueueSender for a Remote Queue, but not a QueueBrowser or QueueReceiver.

Remote Server The actual server at the other end of a routing connection. SonicMQ log messages typically refer to a Remote Server by showing the routing node name and the server name, separated by a colon, for example `Portal:serverA`.

Reply A message sent in response to a request.

Request A message expecting a response. Requests contain a `JMSreplyTo` value in the header, specifying a destination to which a reply to that message should be sent. If the JMSreplyTo value is null, no reply is expected.

Retrieve Threshold When the total size of messages, in KB, stored in memory is less than or equal to Retrieve Threshold, messages in the database are downloaded into memory, subject to the restriction that there can be no more than Save Threshold KB of messages in memory. See *Save Threshold*.

Rollback In a transacted session, destroys the series of messages staged since the last call without sending them. In a session that is both sending and receiving, a rollback destroys the produced messages staged since the last call, and redelivers the series of consumed messages retained since the last call.

Route Table A table used to dynamically maintain information on global queues for routing purposes. It allows the routing queue to determine where messages should be sent during routing.

Route Table Forwarder (RTF) The propagation of routing information is handled by the route table forwarder (RTF) and is referred to as advertising. The RTF accepts route information from other servers in the system, and forwards this information to other servers. The RTF is responsible for updating the information in the Route Table as informational messages are processed. The RTF also obtains current route information from neighboring servers when the routing system is initialized. The RTF needs access to the logical connections in a server.

Router Either a separate piece of hardware or a piece of software running on a host machine that decides where to send packets that are directed to it. The packets travel from router to router until they reach their destination. Each router in the network seeks to answer one question: "Where do I send this packet?" Each router interfaces with other routers to find out the answer.

Routing A routing definition includes the following information for a node:

- A list of connection URLs
- A username associated with the routing
- A flag which indicates whether the node allows connections URLs to servers in the node to be reset for load balancing
- A flag which specifies whether connection URLs to servers in the node are made sequentially or randomly
- A connection timeout
- A flag which specifies whether advertising is turned on or off
- A flag which specifies whether the routing is static or dynamic

The SonicMQ administrative tools set the routing information for a routing node. See *Username, Advertising, Static Routing, Dynamic Routing*.

Routing Connection Routing connections map a given routing node name to a list of possible connection URLs (for example, `ssl://marketplace.progress.com:2506`). When a routing connection to a routing node is required, a server cluster connection is made using one of these connection URLs. When a routing connection is active, then the active connection is used and the connection URL list is not examined.

Routing Connection Table For pre-configured connections, a table of connection routing information called the routing connection table is stored with the configuration database. In a clustered configuration the routing connection table is centrally administered in the configuration server. It defines the connection parameters and options used to establish new connections to a given routing node, if no active connections exist.

Routing Node A message server (standalone or as a member of a cluster) that is configured to allow server-to-server queue routing. A routing node has an associated routing node name shared by all servers in the routing node.

Routing Node Name The name used to identify a routing node. When two servers connect for queue routing, each can advertise the list of remote queues supported. You can configure servers to disable this routing.

Routing Queue A SonicMQ system queue that the message server populates with messages that are destined for a remote queue. The routing queue, which is automatically created on servers in a routing node, stores the messages until a routing connection is established and the messages are forwarded to the remote routing node. The defaule routing queue name is `SonicMQ.routingQueue`.

Routing User A special user stored in a SonicMQ security database that is authorized to connect server-to-server for routing purposes. When a server has security enabled, all connections from other **routing nodes** must identify themselves with a valid routing user name. A routing user has one and only one associated **routing node name** by which this user can by identified.

S

Save Threshold The maximum total size of messages, in KB, that can reside in memory at one time. As additional messages are sent to the queue, they are saved in the database. See *Retrieve Threshold*.

Scalability Used to describe an architecture where you can effectively add resources as traffic increases.

Secure Socket Layer (SSL) A protocol that enables encrypted, authenticated communications across the Internet.

In an SSL connection, each side of the connection must have a Security Certificate, which each side's software sends to the other. Each side then encrypts what it sends using information from both its own and the other side's Certificate, ensuring that:

- Only the intended recipient can decrypt it.
- The other side can be sure the data came from the place it claims to have come from.
- No one has tampered with the message.

Security Administrator A person whose job it is to administer security. The SonicMQ Security Administrator manages security by using the SonicMQ Explorer, the Admin Tool, or the Management API.

Security Policy A set of rules that defines access to and from a corporate network. A security policy must balance the risks and benefits of distributed information and establish acceptable guidelines for employee behavior. A security policy often limits the freedom that external (typically Internet) users have to corporate data and limits the access that internal users have to corporate or external sources of data.

Sender A message producer in Point-to-point messaging. The sender sets headings, properties, and body content for the messages, then produces the messages to destinations maintained by the message server.

Sequential List Access A method for accessing a connection URL list. The first server in the list will be tried first. Sequential start is simple and works well for most applications.

Server See *Configuration Server, Message Server.*

Session A JMS session represents a single thread of activity. All messaging is done through a session object. Each message handler is associated with a single session; a session can have multiple message handlers. The session interface is determined by the messaging paradigm chosen for the connection (either Pub/Sub or PTP).

Single Message Acknowledge See *Acknowledgement Modes*.

SSL Acronym for *Secure Sockets Layer.*

Static Routing For a static routing connection, a remote server connecting with the routing node always establishes a connection to a server that you specify.

Stream Message A stream of Java unkeyed primitive values that is filled and accessed sequentially.

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

Subscription Name The arbitrary string name that identifies a durable subscription and its message selector string. Typically used as a title for the message selector string, perhaps on a server.

Synchronous Delivery See *Delivery Method*.

T

Template Character See *Wildcard*.

Text Message A message containing a `java.lang.String` or **`String`**.

Time to Live (TTL) A long value set by the message producer that, when added to the timestamp from the client system, determines the expiration time of the message. The time to live value is set in milliseconds, and can be any positive integer. A value of zero retains the message indefinitely.

Topic Objects that provide the publisher, server, and subscriber with a destination for JMS methods. Topics can be static objects under administrative control, dynamic objects created as needed, or temporary objects created for very limited use. The topic name is a string of any `javax.jms.String` length. See *Unbound Topic*.

Topic Hierarchy See *Hierarchical Namespace*.

TopicConnection Factory An administered object used to create a connection that binds a Pub/Sub client to a message server at runtime.

Trading Partner An organization that communicates with other trading partners through a portal.

Transacted Session Combines a group of one or more messages with client-to-server ACID properties: Atomic, Consistent, Isolated, and Durable.

When a session is transacted, message input and output are staged on the message server system but not completed until you call the method to complete the transaction; either **`Commit`** or **`Rollback`**. See *Commit, Rollback*.

Triple DES An enhancement to DES that provides considerably more security than standard DES, which uses only one 56-bit key. Triple DES encrypts a message three times. There are several triple DES methods. See *DES*.

U

Unattended Installation An installation that presents no GUI and can be run unattended, so you can use it for remote installations or in a script so that many installations can be performed without user intervention. Also called batch or silent installation.

Unbound Destination A null destination name that, for example, establishes the `QueueRequestor` to bind to that queue space. Often referenced as an unbound topic or an unbound queue. See *Topic* or *Queue*.

Unicode An international 16-bit character code standard.

Unroutable Message A message that arrives at a routing queue where there is no information about the routing connection specified.

User Name Defines an entity's identity. Maintained by the SonicMQ server's security database to authenticate a user with the SonicMQ server and to establish privileges and access rights.

V

View A feature of the Metrics node of the SonicMQ Explorer which provide a dynamic graphical plot of a metric value over a selected time span.

W

Wildcard One of the characters asterisk (*) or pound sign (#) that have special meaning in topic or queue names. A wildcard stands for an arbitrary string, subject to restrictions. With a wildcard, a client can subscribe at once to any number of topics whose name has a given form. Wildcards also enable an administrator to assign permissions to all topics or queues of a given form. Wildcards are also known as template characters.

X

XML Message A message type that contains tagged text that intends to be well-formed XML that can be parsed into a Document Object Model.

Index

P

Q

R

S

T

U

W

X