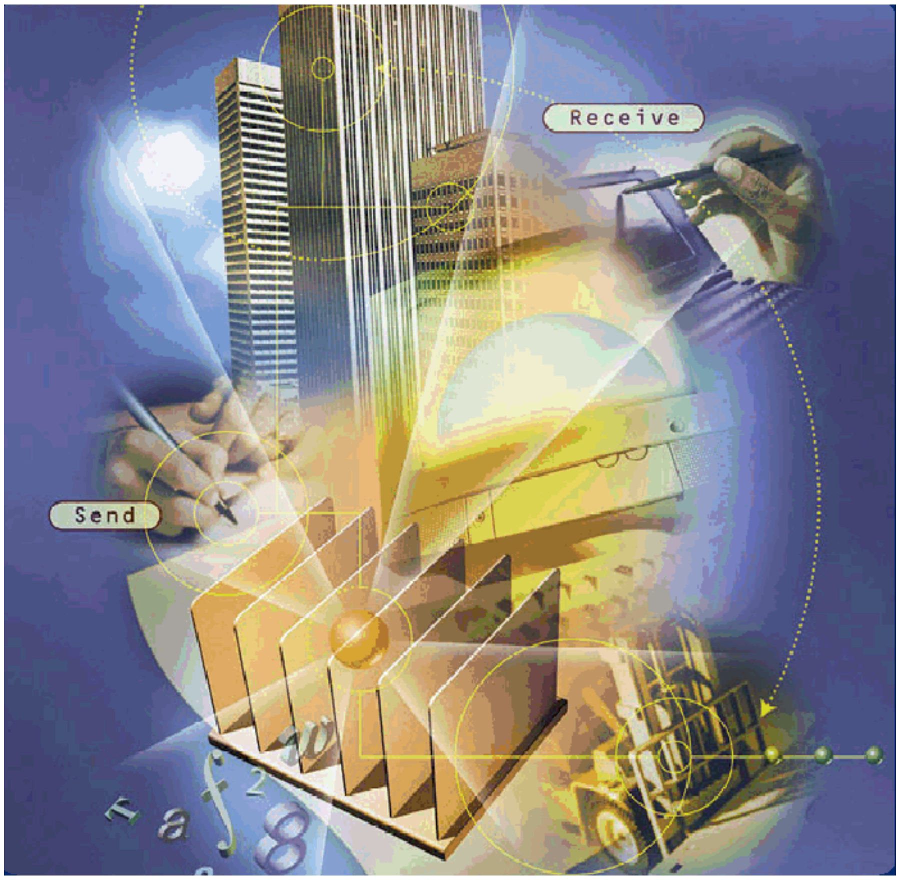
Progress®

SonicMQ™

SonicMQ Deployment Guide



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 specification Version 1.0.2, an API for accessing enterprise messaging systems from Java programs.

This book is divided into two parts. The first part deals with issues you should consider when planning your SonicMQ deployment and consists of the following chapters:

- Chapter 1, "Types of Deployments," discusses the capabilities and limitations of single-server and cluster configurations in terms of performance, scalability, and reliability.

- Chapter 2, "Multi-node Architecture," describes how the Dynamic Routing Architecture and other concepts apply when implementing a portal and trading partner marketplace.
- Chapter 3, "Guaranteeing Messages," describes the use of the Dead Message Queue and the handling of undeliverable messages.
- Chapter 4, "Failover and Load Balancing," discusses how connect-time failover lets a client (or server acting as a client) connect to any server in a user-supplied list. This chapter also dicusses the load-balancing feature, which lets a client (or server acting as a client) be redirected to another server for the purpose of redistributing load.
- Chapter 5, "Security," presents an overview of how to plan and implement a secure SonicMQ installation and explains how to use Signed Applets.
- Chapter 6, "Designing Messaging Models," gives a conceptual overview of how the portal and trading partner B2B application can be used to enable various business-to-business scenarios.

The second part of the book describes how to implement your SonicMQ deployment and consists of the following chapters:

- Chapter 7, "Dynamic Routing Architecture in a Multi-node Application," describes key elements of the Global Queue Routing Architecture in terms of a marketplace application.
- Chapter 8, "Implementing Multi-node Installations," describes the steps you might follow to set up a SonicMQ deployment with portals and trading partners.
- Chapter 9, "Running a Sample Multi-node Application with the Dynamic Routing Architecture," gives step-by-step instructions on how to set up a demonstration portal and trading partner.
- Appendix A, "Performance Tuning," discusses how you can tune some parameters of your SonicMQ configuration to optimize the overall performance of your implementation.

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 preceding 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 especially needed to understand the concept or procedure being discussed.

Important An **Important** flag indicates information that must be acted upon within the given context in order 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.

Table 1. The SonicMQ Documentation Set (*continued*)

Document	*Description*
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`

Table 2. Progress Software International Offices (*continued*)

Locale, Office Name, and Address	*Contact Information*
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`

Part I

Planning Your Deployment

Part I of the *SonicMQ Deployment Guide* deals with issues you must consider when planning your deployment and contains the following chapters:

- Chapter 1, “Types of Deployments,” discusses the capabilities and limitations of single-server and cluster configurations in terms of performance, scalability, and reliability.
- Chapter 2, “Multi-node Architecture,” describes how you can use Dynamic Routing Architecture to implement a B2B deployment, such as a portal and trading partner application.
- Chapter 3, “Guaranteeing Messages,” describes the use of the Dead Message Queue and the handling of undeliverable messages in a multi-node deployment.
- Chapter 4, “Failover and Load Balancing,” discusses how connect-time failover lets a client (or server acting as a client) connect to any server in a user-supplied list, so a connection can be made even if some of the servers in the list are not available. This chapter also dicusses the load-balancing feature, which lets a client (or server acting as a client) be redirected to another server for the purpose of redistributing load.
- Chapter 5, “Security,” presents an overview of how to plan and implement a secure SonicMQ installation and explains how to use Signed Applets.

Chapter 1 Types of Deployments

This chapter consists of several sections:

- "Single-server Configurations" on page 21 briefly sets forth the capabilities and limitations of single-server configurations in terms of performance, scalability, and reliability.
- "Multi-server Clusters" on page 22 describes the performance, scalability, and reliability advantages of clusters.
- "Multi-node Configurations" on page 23 reveals the limitations of using a multi-server cluster in certain types of applications, and briefly describes how these limitations can be overcome by using a multi-node configuration.

Single-server Configurations

A single-server configuration is fine for development and initial testing, but for production use it suffers from two main limitations:

- **Scalability is limited by the capacities of the host machine —** Many contemporary commercial applications must send and receive data from more clients than can be handled by a single computer.
- **Availability is limited —** If the success of your business depends upon critical applications being available 24 hours a day and 7 days a week, your messaging system must be able to work if any single machine goes down. If a messaging client loses a connection to a particular machine, it might be essential that it can still use the messaging infrastructure.

Note Throughout this book, the terms “broker” and “server” will be used interchangeably.

Multi-server Clusters

The requirements of performance and availability can largely be met by using multi-server clusters, supported by SonicMQ Developer, Professional Developers, and E-Business Editions. As explained in the *SonicMQ Installation and Administration Guide*, a cluster consists of a group of interconnected servers. You centrally administer the cluster using a configuration server, which can be part of the cluster, but does not have to be.

Clusters and Scalability

Clustering allows performance to be scaled by adding additional servers to handle heavy message loads. SonicMQ provides the option of using a round-robin algorithm to assign connections so that all servers in a cluster share the load. The following section discusses some issues you should consider when adding additional servers.

Multi-CPU Machines

The servers in a cluster need not be on different machines. You can use a multi-CPU machine, with each of several servers running on its own CPU and its own JVM instance. However, this adds complexity to the installation and might not be faster than using one server. A single server makes effective use of multiple CPUs. In stress tests against a single server on a four-CPU machine, all four CPUs attained close to 100% utilization. Less stressful tests also showed a fairly even load distribution across the four CPUs.

Whether to use a single multi-CPU machine or multiple single-CPU machines depends on several factors:

- On a multi-CPU machine, if you are using one server or if all servers share a single database and the database can be put on the same machine, using a multi-CPU machine should reduce disk access time. In this situation, the multi-CPU solution would be faster.

- A multi-CPU machine is likely to have a single I/O controller, so multiple servers on such a machine would be competing for disk access, making the multi-CPU solution slower.
- If all servers are on a multi-CPU machine and the machine fails, the messaging system will be unavailable. However, if the servers are on individual machines and one fails, parts of the messaging system remain available.

Clusters and Availability

If a SonicMQ client loses its connection to a server or the server fails, the client can redirect its messages to another server in the cluster and can receive messages from other servers.

When the server or network connection comes back up, information can be sent from that server to the one for which the message was originally intended. Alternatively, you can design your application so two servers in a cluster are mirror images of one another. If you do this, your applications will be able to reconnect and continue operation if a single server fails.

If the configuration server goes down, you lose the ability to administer security and cluster membership, and customers lose the ability to administer the routing connection table and routing users that are part of the Dynamic Routing Architecture. In this case, however, existing connections between the servers are maintained, and work can continue uninterrupted.

Multi-node Configurations

Although clusters solve many of the problems you might otherwise encounter when using a single server, there are certain situations where clusters by themselves are not sufficient. These situations are ones impacted by cluster size limitations and cluster functionality limitations, which are discussed in the following sections.

Cluster Size Limitations

Every server in a cluster must maintain a connection to every other server. This means that an n-server cluster must have n * (n - 1) / 2 interserver connections.

Thus, a 16-server cluster has 120 interserver connections and a 32-server cluster has 496 interserver connections. The overhead in maintaining large numbers of connections is excessive. For this reason, you should have no more than 16 servers in a cluster. For large scale applications requiring more than 16 servers another solution is required. SonicMQ technology provides a solution.

Cluster Functionality Limitations

Clusters have functional limitations as well. Some of the very features that let clusters work well within a single enterprise become problems when applications are spread over more than one enterprise:

- Clusters have centralized security management, a valuable convenience within an enterprise. However, an inter-enterprise solution must have local management. Each enterprise must have **local management** to control access using a locally maintained list of access control rights.
- Applications in one enterprise must be able to work even when the remote enterprise is unavailable. That is, they need a store-and-forward capability to allow **disconnected operation**.
- Connections must be secure at the enterprise level, not at the level of the ultimate user. One enterprise generally will have no way to authorize and authenticate individual users in the other enterprise. It must be possible to enforce **enterprise-level connection security**.
- You might want some sites to act as an intermediary. The main function of these sites is to look at the message envelopes and business relationships between two or more external enterprises and forward messages to the appropriate site. This is complex routing based on business logic that cannot be performed by a standard JMS implementation.
- As traffic increases, you must add resources to handle services and routing applications. The architecture must be **scalable**.

The Dynamic Routing Architecture Solution

The SonicMQ solution to the shortcomings of the cluster approach is called **Dynamic Routing Architecture (DRA)**. In this approach, each cluster of servers or unclustered server is a **node**. The DRA is a multi-node architecture. It provides a way to send messages from a server on one node to a destination on another node. Figure 1 shows a message being sent from a client on one node to a client on another node.

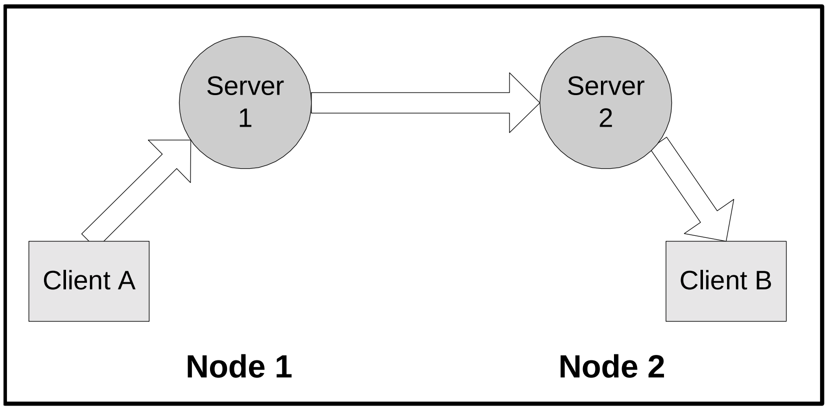


Figure 1. Intra-node Messaging

Within the DRA, SonicMQ serves as the underlying transport layer for messages. SonicMQ gives you the deployment options for physical transport (for example, SSL or TCP) as well as Quality of Service options for guaranteed messages.

Practically speaking, this means that each node has a server and messaging infrastructure. This server communicates to the cluster through a **Routing Queue** built on standard SonicMQ inter-server messaging.

Back-end scalability is achieved by using this Routing Queue to direct messages between nodes. Availability and scalability are attained by allowing the routing queue to load balance queue forwarding across connections from applications that share replicated functionality.

Most of the remainder of this manual explains the ramifications of Dynamic Routing Architecture. In particular, you will need to learn about the topics listed in Table 3.

Table 3. Dynamic Routing Architecture Topics

Topic	*Section or Chapter Where Discussed*
Routing Nodes	"Routing Nodes" on page 30
Queue Routing	"Store & Forward Queue Routing from a Trading Partner" on page 110 "Queue Routing from Portal to Trading Partners" on page 125
Configuration	"System Management" on page 127 "Trading Partner Configuration" on page 136 "Portal Configuration" on page 142
Load Balancing	Chapter 4, "Failover and Load Balancing" "Load-balanced Trading Partner Connections" on page 113 "Load-balancing Across Portal Applications" on page 122
System Management	"System Management" on page 127 "Portal Management" on page 127 "Trading Partner Management" on page 127
Dead Message Queue	Chapter 3, "Guaranteeing Messages" "Dead Message Queue" on page 128

Chapter 2

Multi-node Architecture

Sonic MQ provides a complete, robust implementation of the Java Message Service (JMS). JMS provides reliable, secure guaranteed messaging between applications over the Internet. However, JMS by itself operates at too low a level to represent the complexities of business-to-business E-commerce. For example, imagine thousands of companies working together as trading partners through a marketplace, implemented as a portal. This scenario and other business-to-business E-commerce scenarios are discussed in Chapter 6, "Designing Messaging Models."

In the portal and trading partner scenario, the trading partner plays the role of a messaging client, and the portal acts as a message server that performs routing, logging, authentication, and other services.

This chapter contains two sections:

- "Global Messaging Scalability and Routing Nodes" on page 28 describes the routing node concept at a very high level.
- "Dynamic Routing Architecture" on page 29 describes, at a relatively high level, how routing nodes and the various other components of the architecture work together to allow the implementation of multi-node solutions.

Global Messaging Scalability and Routing Nodes

You can implement SonicMQ messaging either on a single server, or on a single cluster of servers. Each of these acts as a node for messaging where configuration can be centrally administered and where clients and servers can be fully connected using normal JMS semantics.

In the global, business-to-business case, there is a need for connecting these isolated, independently administered messaging nodes. SonicMQ's Dynamic Routing Architecture lets you connect these messaging nodes through the concept of SonicMQ routing nodes.

You can define any SonicMQ server, or cluster of servers, as a routing node. Network connections can be configured at any routing node to any list of other **adjacent** routing nodes. Adjacent routing nodes are those nodes that have a server-to-server connection with the current node. This connection can be active, or you can define it administratively. This allows any client at one routing node to address messages to both local queues and to remote queues on adjacent routing nodes.

SonicMQ will deliver messages to adjacent routing nodes with its commitment of guaranteed, exactly-once delivery. You can add additional routing at the application level to take messages and send them on to subsequent routing nodes. Figure 2 illustrates this concept.

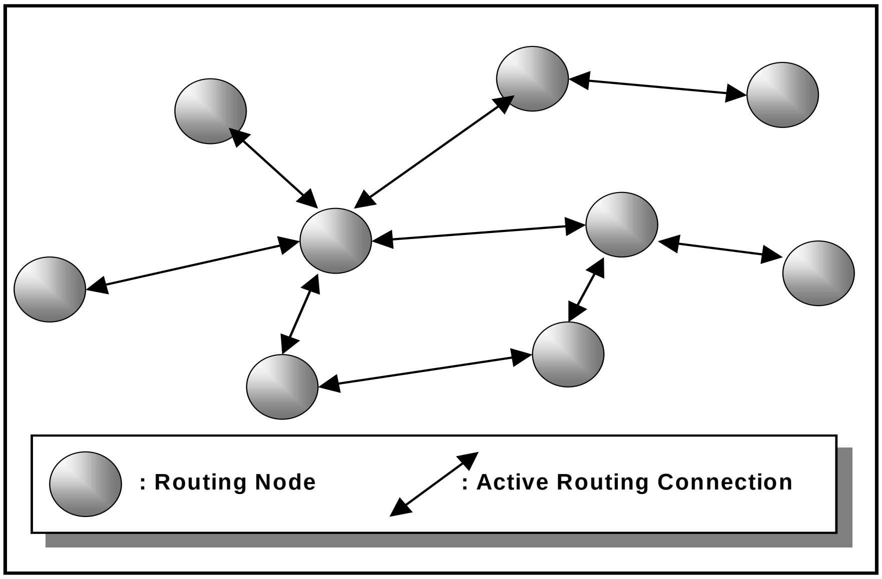


Figure 2. Routing Nodes

Each of the routing nodes in Figure 2 represents either a single server or a cluster, plus all of the clients directly connected to the servers in that node.

Dynamic Routing Architecture

The ability to scale to thousands of trading partners and many portal applications and services is based on the extension of SonicMQ inter-server messaging. This messaging has been enhanced to support global queue routing between routing nodes. This global queue routing structure is part of the SonicMQ **Dynamic Routing Architecture**.

Routing Nodes

With global queue routing, it is possible to send messages to global queues that reside on other servers. You do this by creating a routing node.

A **routing node** can be a single unclustered server or a cluster of servers. Routing node names are only exposed to adjacent routing nodes in a network of routing nodes. Adjacent nodes can be established dynamically (as a remote routing node connects into a node) or preconfigured in the server or configuration server database (for outgoing connections).

For JMS clients connected to a server or cluster, you can specify queues that exist in adjacent routing nodes by prepending the name of the routing node to the queue. A queue name that is qualified by a routing node name is referred to as a **remote queue**. The syntax for a remote queue is *`routing_node_name`*::*`queue_name`*. JMS clients can retrieve messages only from queues in servers to which they have established connections.

For example, a JMS client at a trading partner can send to the **`appQ`** queue on the **`Portal`** routing node by using the remote queue name `Portal::appQ`.

A routing node name can be any Java string of up to 256 Unicode characters that does not contain a double colon (**`::`**) or double quote (**`"`**) and does not begin with a dollar sign (**`$`**).

The following client code takes advantage of remote queue naming:

```
// Create the QueueSender on the Queue
javax.jms.Queue remoteQueue = session.createQueue (“Portal::appQ”);
javax.jms.QueueSender qSender = session.createSender(remoteQueue);
// Send a Message
qSender.send(msg);
```

The use of global queues is subject to the following rules:

- You cannot create a QueueReceiver or a QueueBrowser on a remote queue. Attempting to do so will raise an exception.
- The remote queue must exist on the destination routing node. Existence of the remote queue is checked only at the destination routing node. If the queue does not exist, the message is flagged as undeliverable and, if the sender has so requested, moved to the **`SonicMQ.deadMessage`** queue.
- Access control by a client to a remote queue is based on the queue name, without the routing node name. That is, the ACLs are checked at the server connected to the client.

- Remote queues must be defined as **global** on the destination routing node to have routing. Global queues are advertised over new routing connections, unless you turn off this capability.
- To send messages to a global queue on a local routing node, you can omit the routing node name and simply preface the name with a double colon (**`::`**), as in this code:

  ```
  session.createQueue (“::appQ”);
  ```

  You can omit the routing node name for a global queue that exists on the server that the JMS client is connected to, or to a global queue that is defined on another server in the same cluster.

Behavior of a Routing Queue in a SonicMQ Server

The name of the routing queue is `SonicMQ.routingQueue`.

The routing queue automatically routes all messages that are not local to that routing node to remote queues. The connection associated with a routing node is initialized administratively. That is, each routing node name has an associated **connection** (which maps to a list of server URLs, ports, and other connection parameters).

Note Routing is enabled for all servers.

When two servers connect to each other for queue routing, they exchange information on queues that are explicitly set **global**. This is referred to as the **advertising** of global queues. You can disable advertising by selecting the **`noadvertise`** flag when setting a routing in the Admin tool or SonicMQ Explorer.

Messages are checked at arrival at a routing node for user/queue write permissions based on the security ACL configuration at the receiving server.

You specify the name of the routing node by adding the following line to the `broker.ini` file:

```
ROUTING_NODE_NAME=name
```

You cannot delete or rename the routing queue. However, you can modify its properties such as maximum size, save threshold, and retrieve threshold by using the Admin tool, Explorer, or the Management API.

A **route table** is 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. When a global queue is advertised from a routing node, the table retains the connection information associated with that queue. Only the most current information is retained, along with the shortest path to the destination queue.

The information in the route table is persisted as it is received so that remote queue routings are known at server startup, even if no routing advertisements are received. The full connection information for a destination queue is retained, which allows outgoing connections to be established, if possible.

For preconfigured 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. Figure 3 illustrates the configuration of the routing connection table.

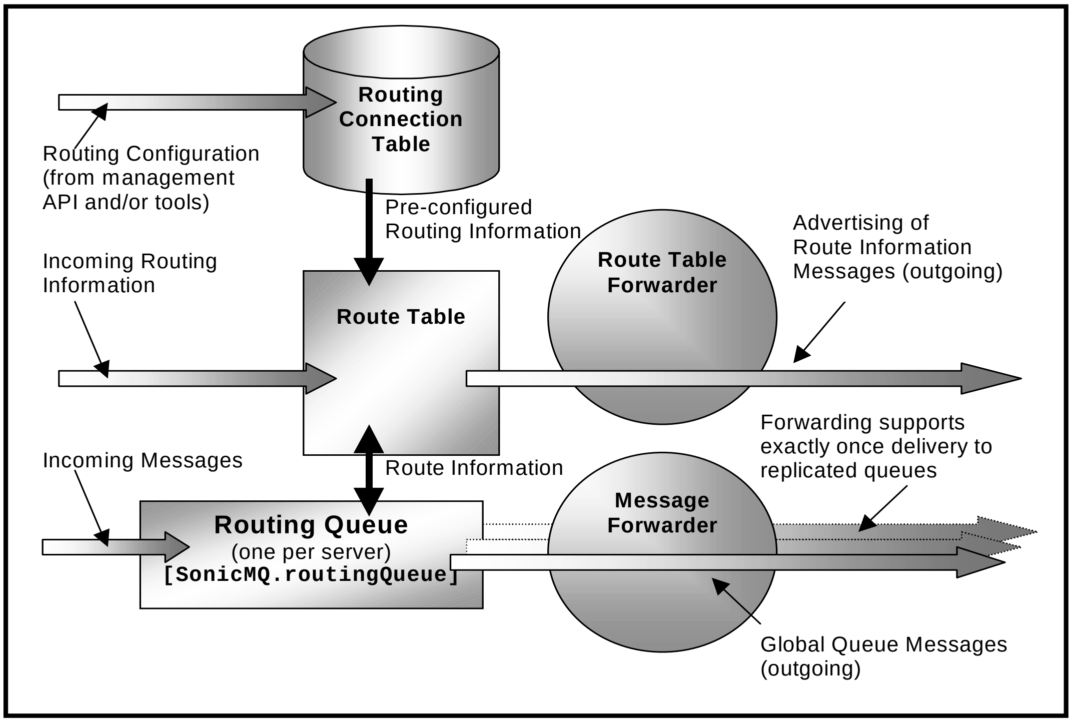

Routing Connection Table
Routing Configuration (from management API and/or tools)
Pre-configured Routing Information
Incoming Routing Information
Route Table
Route Table Forwarder
Advertising of Route Information Messages (outgoing)
Incoming Messages
Route Information
Forwarding supports exactly once delivery to replicated queues
Routing Queue
(one per server)
[SonicMQ.routingQueue]
Message Forwarder
Global Queue Messages (outgoing)

Figure 3. Routing Connection Table

Configured and Advertised Routing Information

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.

The following restrictions apply to the advertising of queues when servers are connected:

- Only queues explicitly defined as global are advertised.
- Only global queues defined on a routing node are advertised to another routing node.
- When servers make connections from one routing node to another, the connection can be configured to explicitly prevent this advertising. A global queue is not advertised through a routing connection that disables advertising.
- Within a cluster, the advertising of global queues always happens. This is automatic and occurs when the server is added to the cluster or when new global queues are created or deleted.
- Advertising must be explicitly turned on for routing connections defined administratively between nodes.
- Any new routing information received by a clustered server is immediately propagated to servers in that cluster. The new information will be advertised to adjacent nodes only if that information pertains to the node itself.
- The server originating routing information will include a timestamp to allow for duplicate updates to be detected. Only the most recent information will be used.
- Duplicate routing information is not forwarded. This prevents advertising to enter an infinite loop in complex routing configurations.

Routing Nodes and Clusters

A routing node can consist of a single server or a cluster of servers. If the node is a cluster, static routing connection information is configured for the entire cluster through the configuration server. That is, the relationship between the routing node name and outgoing connections is set for the entire cluster, and must be set on the configuration server. This is similar to the way that security information is administered and stored. Routing and security configurations are designed to work together, but neither is a prerequisite for the other. You can configure a cluster for routing, for security, or for both.

Changes in route-table information are shared between servers in the routing node using internal messaging. This applies to changes made administratively though the management tools or API, as well as to routings defined dynamically due to route table advertising.

Routing connections within a cluster-based routing node are made automatically. Each server in a cluster-based routing node can route to any other server in the same routing node. That is, within a cluster, routing connections are, at most, one forwarding “hop.”

Using a cluster does not automatically create a routing node. To create a cluster that is also a routing node, every server in the cluster must be configured with the same values in the individual `broker.ini` files. For example, to set up the Portal routing node, you set `ROUTING_NODE_NAME=Portal` for each server. Each of the servers in the cluster is added to the cluster using the configuration server.

The configuration server can also have the same settings, but only if it is to be added to the cluster itself. A configuration server does not have to be part of the routing node in order for it to administer routing connection information.

Chapter 3 Guaranteeing Messages

This chapter provides information about how you can use the SonicMQ Dead Message Queue (DMQ) features to guarantee that messages will not be discarded until a client has processed them. The chapter contains the following sections:

- "Working with Dead Message Queues" on page 37 describes dead messages and dead message queues, and the ways SonicMQ provides for you to handle them.
- "The System Dead Message Queue" on page 41 provides information about the properties of the SonicMQ system dead queue.
- "Handling Undelivered Messages" on page 44 describes the process SonicMQ uses to handle undeliverable messages
- "Types of Undelivered Messages" on page 47 defines the various cases where messages are marked as undelivered and provides reason codes and descriptions of each type of undelivered message, including the scenarios in which the undelivered message might occur

Working with Dead Message Queues

JMS provides mechanisms for guaranteed delivery of messages between clients and within the provider. However, there are cases where messages are allowed to expire or where they are viewed by the provider as undeliverable. These messages are called **dead messages**.

If you have a local application of SonicMQ, the only dead messages you should encounter are those that expire. The other types of dead messages discussed in this chapter arise in multi-node deployments. These deployments are discussed in Part II, "Implementing Your Deployment."

Note The DMQ is used only for messages delivered in the Point-to-Point domain.

SonicMQ provides you with the ability either to deal with these messages or to be kept aware of situations where messages are not being delivered due to high latency or possible provider failure. This ability is achieved through use of the Dead Message Queue.

When you use the SonicMQ Dead Message features, the SonicMQ server will deal with undeliverable messages as follows. When the SonicMQ server finds messages that have exceeded their time to live (TTL) and should expire or that cannot be routed due to some external network error, the server:

- Saves the message in a **dead message queue** (**DMQ**)

  and/or

- Generates an **administrative notification (**management event**)**

At an application level, you can listen for the administrative notifications, browse the DMQ, and deal with undelivered messages as appropriate for your application. For more information about performing these tasks, see the *SonicMQ Programming Guide.*

Note Messages sent with a NON_PERSISTENT delivery mode are subject to a lower quality of service than PERSISTENT messages. NON_PERSISTENT messages in the DMQ are not retained after a planned or unplanned shutdown of the server. These messages must be processed in the same server session in which they occur, otherwise they will be discarded.

What Is an Undeliverable Message?

In the case of server-to-server queue routing across routing nodes, there are cases where messages are considered undeliverable. (Part II, "Implementing Your Deployment," introduces and discusses dynamic routing architecture.) These cases include the following types of messages:

- **Unroutable messages** are messages that arrive at a routing queue where the information on the routing is missing or incomplete.
- **Indoubt messages** are messages that have been forwarded to another routing node, but where the handshaking needed to ensure once-and-only-once delivery of messages has been interrupted due to network or hardware failure and cannot be re-established within the cofigurable INDOUBT_TIMEOUT (approximately one business day).

There are other reasons why a message might not be delivered, including timeouts and network failures. See the "Types of Undelivered Messages" section for descriptions of various scenarios under which messages are not delivered.

Messages that do not make forward progress during queue routing for a configured period of time are transferred to the DMQ. This period of time is specified by the TTL parameter.

Using the System Dead Message Queue

In SonicMQ, all undeliverable messages are sent to the system DMQ, named `SonicMQ.deadMessage`. The system dead message queue is treated exactly like a normal queue in that it can be browsed or read using normal JMS objects (QueueBrowser and QueueReceiver). The only special handling feature of these queues is that messages are not allowed to expire from them.

Guaranteeing Delivery

JMS can guarantee delivery by using queues and setting the delivery mode to PERSISTENT, but cannot guarantee latency of messages. When you use the DMQ, any expired message is guaranteed to be preserved on the server. To ensure that expired messages are preserved, you must configure your application to monitor the DMQs and to handle all messages that arrive in the DMQ.

Enabling Dead Message Queue Features

You enable the DMQ features only on a message-by-message basis. You must specifically request enqueuing and notifications of administrative events, or the DMQ is not used. Enabling the DMQ in this way prevents the DMQ from accidentally filling up and shutting down the server.

See the *SonicMQ Programming Guide* for information on setting message properties to request enqueueing on the DMQ.

Monitoring Dead Message Queues

It is very important that your application monitor the dead message queues and deal with messages that arrive there. When any of these system queues exceeds its maximum queue size, the server is shut down.

To help deal with the potential of DMQs filling up, the SonicMQ server monitors the DMQ and sends an administrative event notification when the queue exceeds the predefined percentage. This percentage is set to 85% by default. An event is sent every time a message is enqueued that causes the size of the queue to exceed the notification percentage.

Warning Applications should not directly add messages to the DMQ by creating QueueSenders. Recommended access to the DMQ is through QueueBrowsers and QueueReceivers.

Note Messages are enqueued in the DMQ retain their original destination and JMSExpiration value. Ensure that `QueueBrowsers` and `QueueReceivers` on the DMQ check the `(javax.jms.Message) m.getJMSDestination()` for the original queue. Checking `m.getJMSExpiration()` will always yield a time in the past

The System Dead Message Queue

The Dead Message Queue (DMQs not used unless you request it. You can request administration notifications and enqueuing on the DMQ by setting properties on each message. See the *SonicMQ Programming Guide* for information on setting message properties.

The DMQ is created and populated by SonicMQ. The DMQ has the following properties:

- Exists on every server
- Is created automatically by SonicMQ (all running SonicMQ servers have an active DMQ)
- Is always named: `SonicMQ.deadMessage`
- Is a local queue
- Cannot be deleted

As with other queues, messages that have a JMSDeliveryMode of NON_PERSISTENT are not available in the DMQ after a system shutdown (either planned or unplanned).

As explained in the "Monitoring Dead Message Queues" section, the SonicMQ server will shut down if the DMQ exceeds its configured capacity. Prior to shutting down the server, however, the DMQ will raise an administrative event when it exceeds a fraction of its maximum size. The notification factor defaults to 0.85 (85%). You can reset this value (DMQ_NOTIFY_FACTOR) in the broker.ini file to an appropriate limit for your application. You can monitor these events using the Management API or the tools. SonicMQ raises the event for every message added to the DMQ.

You can configure the SonicMQ Explorer and Admin tool to listen for these messages. For example, in the Admin tool, enter this command:

```
show broker events start dmqstatus
```

See the *SonicMQ Installation and Administration Guide* for more information about using the SonicMQ Explorer and Admin tool.

See the *SonicMQ Programming Guide* for an example that shows you how to use the SonicMQ Explorer to monitor the server's DMQ.

Default DMQ Properties

By default, SonicMQ creates the Dead Message Queue with the properties listed in Table 4.

Table 4. Dead Message Queue Properties

Property	***Value***	***Editable***
Name	`SonicMQ.deadMessage`	No
[local|global]	local	No
[shared|exclusive]	shared	Yes
retrieve threshold	1,200 K	Yes
save threshold	1,400 K	Yes
Maximum queue size	10,000 K	Yes

Modifying Default DMQ Properties

You can modify all the parameters of the `SonicMQ.deadMessage` queue, except the name and local setting, using the Admin tool or Explorer.

The settings for retrieve threshold, save threshold, and maximum queue size are highly specific to an application, therefore you should change these from their default settings to values appropriate to your application.

See the *SonicMQ Installation and Administration Guide* for information about using the Admin Shell and Explorer.

Modifying DMQ Access Control

The administrator can modify Access Control for the DMQ using the Management API, Admin tool, or Explorer. Access Control is set in the same way for the system queues as for nonsystem queues.

See the *SonicMQ Installation and Administration Guide* for information about using the Management API, Admin tool, and Explorer.

JMS_SonicMQ Message Properties

The following is a list of the message properties associated with messages declared undeliverable and possibly moving to the DMQ:

- `JMS_SonicMQ_preserveUndelivered`

  Set this `boolean` property to true for every message that should be transferred to the `SonicMQ.deadMessage` queue when noted as being undeliverable.

- `JMS_SonicMQ_notifyUndelivered`

  Set this `boolean` property to true for every message that should raise an administration notification when noted as being undeliverable.

- `JMS_SonicMQ_undeliveredReasonCode`

  Read this `int` property to determine why SonicMQ declared this message as undeliverable. The server sets this property when messages are moved to a dead message queue.

- `JMS_SonicMQ_undeliveredTimestamp`

  Read this `long` property to determine when SonicMQ declared this message as undeliverable. The server sets this property when messages are moved to a dead message queue.

These property names are available as standard constants in `progress.message.jclient.Constants`. Table 5 provides the values for these constants.

Table 5. JMS SonicMQ Properties

JMS SonicMQ Constant	***String Value***
NOTIFY_UNDELIVERED	“JMS_SonicMQ_notifyUndelivered”
PRESERVE_UNDELIVERED	“JMS_SonicMQ_preserveUndelivered”
UNDELIVERED_REASON_CODE	“JMS_SonicMQ_undeliveredReasonCode”
UNDELIVERED_TIMESTAMP	“JMS_SonicMQ_undeliveredTimestamp”

Handling Undelivered Messages

The following sequence of events describes the process SonicMQ uses to handle undeliverable messages:

1. A condition occurs where the server determines the message is not deliverable. (See the "Types of Undelivered Messages" section for a list of possible causes.)
2. The message is passed to a special processing object in the SonicMQ server. That object examines the message header.
3. The special processing object determines whether to preserve the message in the DMQ:

   The message is checked for the boolean property:

   ```
   JMS_SonicMQ_preserveUndelivered
   ```

   If this property is TRUE, then the message is transferred to the `SonicMQ.deadMessage` queue with the following properties:

   ```
   JMS_SonicMQ_undeliveredReasonCode = reason_code [int]
   JMS_SonicMQ_undeliveredTimestamp = GMT_timestamp [long]
   ```

   See the "Types of Undelivered Messages" section for a description of *`reason_code`*.
4. The special processing object determines whether to send a notification that the message has been sent to the DMQ or that the message has expired:

   The message is checked for the boolean property:

   ```
   JMS_SonicMQ_notifyUndelivered
   ```

   If this property is TRUE, an administration notification is sent with the following information:

   - Reason code
   - MessageID (of the original message)
   - Destination (of the original message)
   - Timestamp (of when the message underwent dead-message handling)

- Name of server (where message originated)
- Preserved boolean (TRUE, if the message was saved to the DMQ)

Programmatic handling of the undelivered message event is done using the Management API calls in `progress.message.tools.BrokerManager`. You must create a class that implements the callback for the `brokerUndeliveredMsgNotification` method. See the javadoc for the `BrokerManager` class and `IBrokerManagerListener` interface in the `progress.message.tools` package for more information on these calls.

Sample Scenarios of Handling Dead Messages

The following sections describe typical scenarios in handling dead messages:

- "Preserving Expired Messages and Throwing an Administration Notification"
- "Using High Priority and Throwing an Administration Notification"

Preserving Expired Messages and Throwing an Administration Notification

Typically, important messages will be sent PERSISTENT and will be flagged both to be preserved on expiration and to throw an administration notification. The following code sample shows how this might be done:

```
// Create a TextMessage for the payload. Make sure the message
// is delivered within 2 hours (7,200,000 milliseconds).
// If expires, send a notification and save the message.
javax.jms.TextMessage msg = session.createTextMessage();
msg.setText("This is a test of notification and DMQ");

// Set 'undelivered' behavior. Optionally, we could have used the
// property names defined as static final Strings in
// progress.messages.jclient.Constants.
msg.setBooleanProperty("JMS_SonicMQ_preserveUndelivered", true);
msg.setBooleanProperty("JMS_SonicMQ_notifyUndelivered", true);
```

```
// Send the message with PERSISTENT, TimeToLive values.
qsender.send(msg,
             javax.jms.DeliveryMode.PERSISTENT,
             javax.jms.Message.DEFAULT_PRIORITY,
             7200000);
```

Using High Priority and Throwing an Administration Notification

The following code would be used to send a small message using high priority, with the expectation that this message will be delivered in ten minutes. In this case, we are only interested in notification events:

```
// Create a BytesMessage for the payload. Make sure the message
// is delivered within 10 minutes (600,000 milliseconds).
// If expires, send a notification.
javax.jms.TextMessage msg = session.createTextMessage();
msg.setText(“Test of undelivered events”);

// Set 'undelivered' behavior. Optionally, we could have used the
// property names defined as static final Strings in
// progress.messages.jclient.Constants.
msg.setBooleanProperty("JMS_SonicMQ_notifyUndelivered", true);

// Send the message for fast delivery, or not at all.
qsender.send(msg,
             javax.jms.DeliveryMode.NON_PERSISTENT,
             8,        // Expedite at a high priority
             600000);  // 10 minutes
```

What to Do When the Dead Message Queue Fills Up

When the DMQ fills up (to its maximum queue size), the server stops processing messages after enqueuing the message that caused the DMQ to exceed its maximum size. In this way, no messages are lost.

If a server shuts down because the DMQ is full, you can restart the server after setting the DMQ_OVERRIDE_MAXSIZE parameter in the `broker.ini` file. The server then starts up with a temporary override on the maximum size of the dead message queue. Assuming the new value is sufficiently large, the queue can be processed or cleared. After the queue is processed, you should restart the server with its original settings.

If, while the DMQ_OVERRIDE_MAXSIZE parameter is in effect, the maximum size of the dead message queue is changed administratively through a tool or the Administration API, this new value is stored in the database and used until the server is shut down. If the DMQ_OVERRIDE_MAXSIZE parameter is removed before restarting the server, the new stored value is used. However, if the DMQ_OVERRIDE_MAXSIZE parameter is left in place, it again overrides the stored value.

See the *SonicMQ Programming Guide* for information about handling dead message queues programmatically.

Types of Undelivered Messages

This section defines the various cases where messages are marked as undelivered. The following sections provide reason codes and descriptions of each type of undelivered message, including the scenarios in which the undelivered message might occur. The types of undelivered messages are:

- TTL is expired
- Routing node is invalid
- Routing destination is invalid
- Connection cannot be established before routing timeout
- INDOUBT_TIMEOUT expires
- Connection authentication fails
- Connection authorization fails
- Message is too large

Other cases where messages might be lost are discussed at the end of this chapter. Some of these types of undelivered messages arise in scenarios involving the dynamic routing architecture; this concept is discussed in Part II, "Implementing Your Deployment."

Note Reason codes are defined as `public final static int` in the `progress.message.jclient.Constants` class.

TTL Is Expired

The reason code is: **`UNDELIVERED_TTL_EXPIRED`**

The SonicMQ server determines that a message has expired.

This dead message event is the simplest case and the one that most developers consider when thinking about dead message queues.

When sending messages, you can optionally set the parameter **time to live (TTL)**. This TTL is converted to an expiration time and is stored in the message header (in GMT).

When a SonicMQ server tries to deliver a message, it notes the expiration time (based on the GMT as calculated from the server's system clock) and might decide not to deliver the message due to expiration.

Checks for expiration are done only periodically within a server (in order to avoid extra overhead). Messages are always guaranteed not to be delivered if they have expired. However, the actual time they are moved to the dead message queue might be significantly later than the expiration date in the header. You can change the QUEUE_CLEANUP_INTERVAL parameter to adjust the frequency of these checks.

Routing Node Is Invalid

The reason code is: **`UNDELIVERED_ROUTING_INVALID_NODE`**

A client tries to send a message to a remote queue for which no routing node connection information exists. Figure 4 shows an example of this situation.

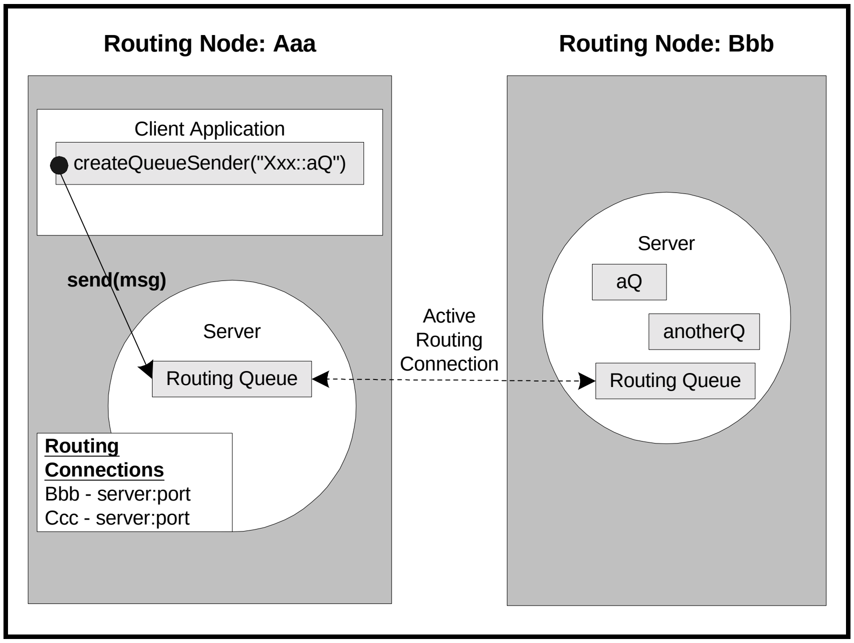


Figure 4. Invalid Routing Node

A client tries to send a message to the remote queue: Xxx::aQ (Routing Node = Xxx; Queue name = aQ).This message goes to the routing queue in the server, Aaa, which is shown to have an active connection with routing node Bbb.

The desired routing node connection, however, is Xxx, which is not active, nor is there default connection information for this node in the routing connection table.

As a result, the message is declared to be undeliverable and the dead message processing will occur. The message will stay on the server at Aaa.

Routing Destination Is Invalid

The reason code is: **`UNDELIVERED_ROUTING_INVALID_DESTINATION`**

A client tries to send a message to a remote queue for which the connection exists, but once the message arrives no global queue is found to exist. (The global queue should exist on the receiving server or on another server in the routing node, if it is comprised of a SonicMQ cluster.) Figure 5 shows an example of this situation.

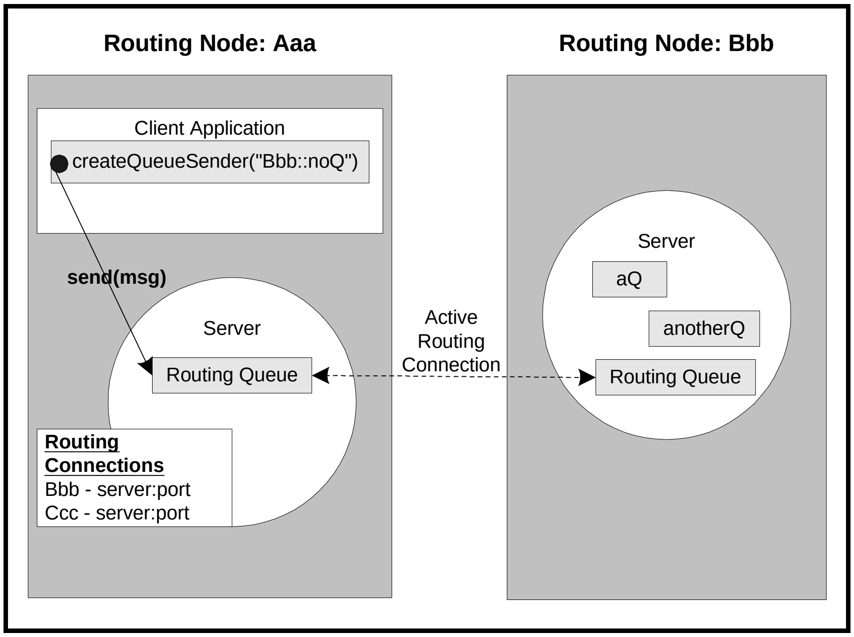


Figure 5. Invalid Routing Destination

A client tries to send a message to the remote queue: Bbb::noQ (Routing Node = Bbb; Queue name = noQ). This message goes to the routing queue in the server, which finds an active connection with routing node, Bbb.

The message is moved to the server at routing node Bbb. When this server tries to deliver the message, however, it realizes that there are no global queues that have this name (including elsewhere in the cluster, if the routing node is clustered).

At this point, the message is sent to the dead message processing logic on the server at Bbb.

Connection Cannot Be Established Before Routing Timeout

The reason code is: **`UNDELIVERED_ROUTING_TIMEOUT`**

A client tries to send a message to a remote queue for which the connection should exist, but cannot be established (or re-established, in the case of a lost connection). Figure 6 shows an example of this situation.

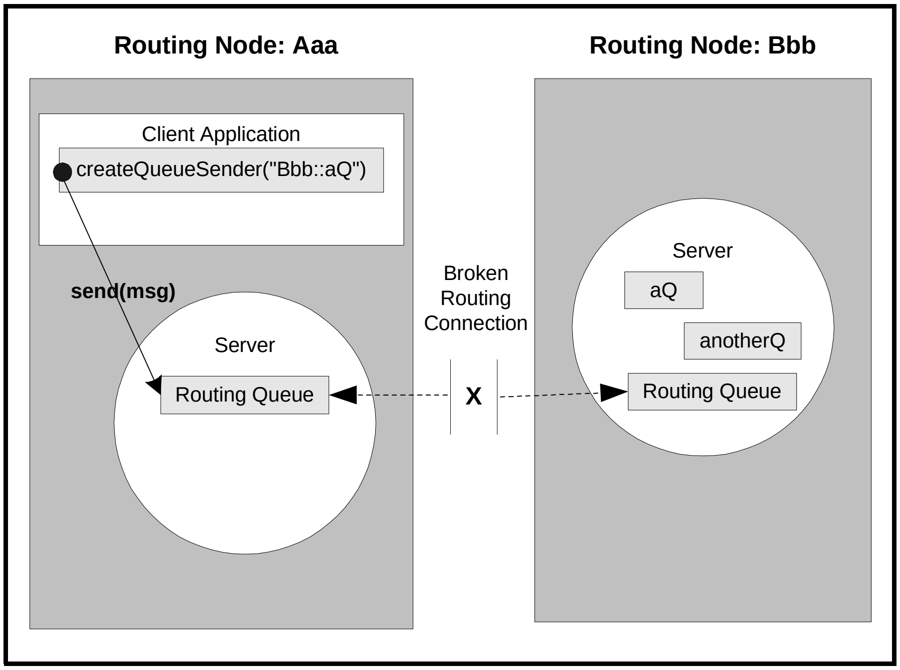


Figure 6. Broken Routing Connection

A client tries to send a message to the remote queue: Bbb::aQ (Routing Node = Bbb; Queue name = aQ). This message goes to the routing queue in the server, which finds a connection with routing node Bbb.

When an attempt is made to use this connection, however, it is found to be down (or perhaps timed out). Repeated attempts to restart this connection to the routing node Bbb fail.

If the failures continue for a configurable length of time (the ROUTING_TIMEOUT setting in broker.ini), the message is sent to the dead message processing logic on the server in routing node Aaa.

INDOUBT_TIMEOUT Expires

The reason code is: **`UNDELIVERED_ROUTING_INDOUBT`**

A network failure or server failure occurs after the sending server has sent a PERSISTENT message, but before it has received an acknowledgement, causing the message to be in an indoubt state. The message remains in this state until a connection is re-established between the two servers (or until the INDOUBT_TIMEOUT expires).

The sending server automatically tries to re-establish any connections necessary to resolve the state of the indoubt messages. Until this occurs, however, all the indoubt messages are held where they will not be lost. There is no possibility of message redelivery due to any failure situation.

SonicMQ handles this situation as follows:

- As part of server configuration, a parameter exists that specifies an INDOUBT_TIMEOUT (in seconds).
- All messages that are in the indoubt state for a period that exceeds this time automatically expire. (Typically, all PERSISTENT messages would be configured to be sent to the DMQ and to raise an administration notification.)
- At no point are these messages lost or inadvertently placed in a state where they can be redelivered.

The important details of this scenario include:

- Messages are never redelivered by SonicMQ queue routing even in the event of network failure.
- Because only PERSISENT messages are subject to the special indoubt handling, only PERSISTENT messages can ever be declared as undeliverable with this reason code.
- Messages may be stored on a sending server in an indoubt state.

- SonicMQ will attempt to re-establish the server-to-server connection to resolve indoubt messages even if another server-to-cluster connection has been created for the destination routing node.
- In the event of an unsuccessful attempt to re-establish a server-to-server connection for the purpose of resolving indoubt messages, SonicMQ will wait the number of seconds specified by the INDOUBT_RECONNECT_INTERVAL parameter before a subsequent attempt is made to re-establish the connection. This cycle is repeated until either the connection is successfully re-established or the routing timeout interval (INDOUBT_TIMEOUT) has passed.
- If the failed connection cannot be re-established, the message is optionally moved to the dead message queue, after the INDOUBT_TIMEOUT. However, there is no reason not to have this parameter set to a long period, as the indoubt resolution process uses the SonicMQ journal to retain state. Even if both servers fail and are restarted in the process, at different times, guaranteed exactly-once delivery is assured.
- Indoubt messages expire only on the original, sending server. A copy is not caused to expire on the receiving server as part of the same network or connection failure. (However, the message might expire later for a different reason, for example, TTL.)

Connection Authentication Fails

The reason code is:
`UNDELIVERED_ROUTING_CONNECTION_AUTHENTICATION_FAILURE`

A message with routing information cannot be delivered to a specified node due to authentication failure (invalid credentials).

The routing node name is valid (that is, it does exist in the routing connections database). However, the routing connection fails because the server being connected to refuses the connection due to invalid credentials. Figure 7 shows an example of this situation.

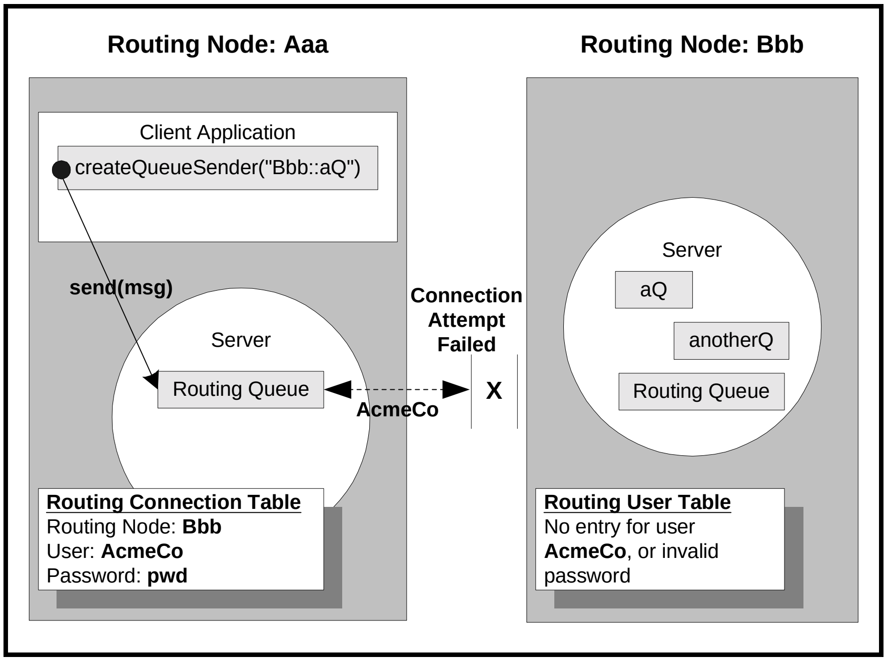


Figure 7. Failed Connection Authentication

A client tries to send a message to the remote queue: Bbb::aQ (Routing Node = Bbb; Queue name = aQ). This message goes to the routing queue in the server for routing node Aaa. This server attempts to create a new connection to routing node Bbb.

The connection information for Bbb is retrieved from the routing connection table at Aaa, which indicates that the connection to Bbb should be done with user=AcmeCo and password=pwd.

The server at routing node Bbb, however, does not have this user/password combination in its table of routing users. The connection is refused, and the message is sent to the dead message processing logic on the server in routing node Aaa.

Connection Authorization Fails

The reason code is: **`UNDELIVERED_ROUTING_CONNECTION_AUTHORIZATION_FAILURE`**

A message with routing information cannot be delivered to a specified node due to authorization failure (insufficient privileges).

The routing node name is valid (that is, it does exist in the routing connections database). A connection could indeed be made, and authenticated, at that routing node. However, the Routing Node Name of the sender does not match the Routing Node Name in the receiver's routing user security database. Figure 8 shows an example of this situation.

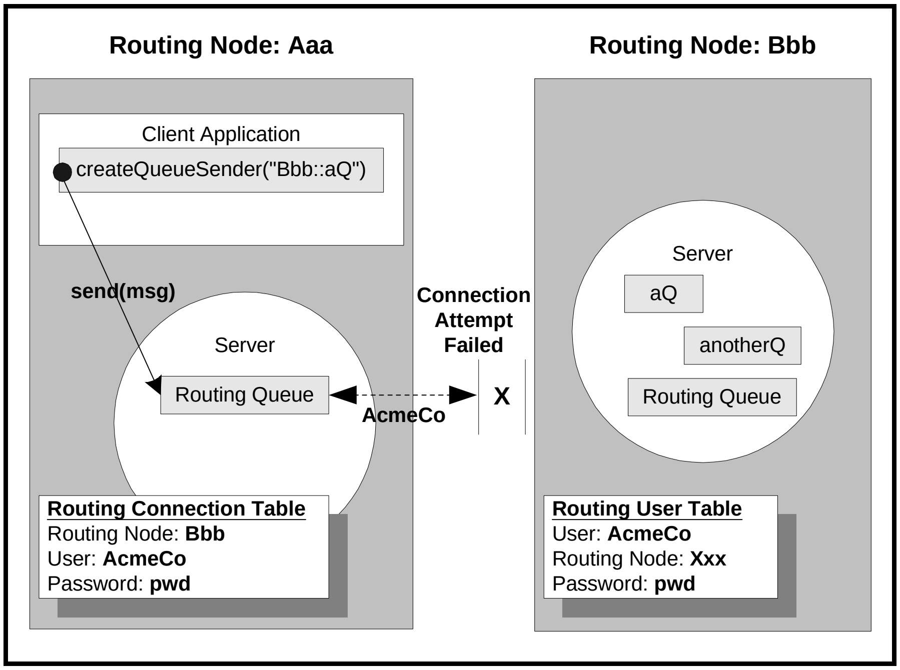


Figure 8. Failed Connection Authorization

A client tries to send a message to the remote queue: Bbb::aQ (Routing Node = Bbb; Queue name = aQ). This message goes to the routing queue in the server for routing node Aaa. This server attempts to create a new connection to routing node Bbb.

The connection information for Bbb is retrieved from the routing connection table at Aaa, which indicates that the connection to Bbb should be done with user=AcmeCo and password=pwd. This connection attempt has the correct credentials, and the server at routing node Bbb does recognize AcmeCo as a valid user with proper credentials.

However, the table of routing users indicates that the associated routing node must be Xxx (and not Aaa). The connection is refused, and the message is sent to the dead message processing logic on the server in routing node Aaa.

Message is Too Large

The reason code is: UNDELIVERED_MESSAGE_TOO_LARGE

An attempt is made to enqueue a message that is larger than the maximum size of a queue.

Normally, an attempt to enqueue a message larger than the maximum queue size would cause an exception to the sender. However, if the sender is another server, as is the case with routing, then the sender cannot catch the JMSException. Instead, the message is sent to the DMQ on the routing server.

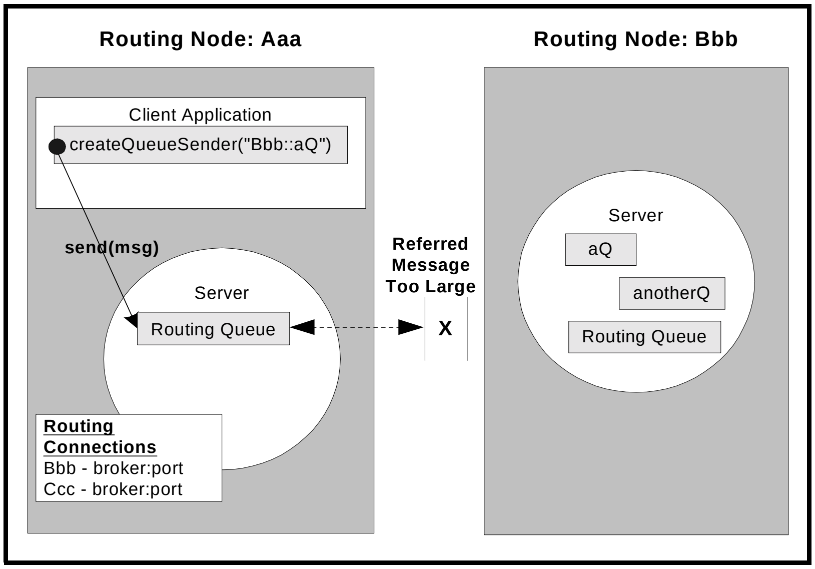


Figure 9. Message is Too Large

In Figure 9 a client tries to send a message to the remote queue: Bbb::aQ (Routing Node = Bbb; Queue name = aQ). However, the message cannot be accepted by Bbb because the message size is bigger than the maximum size of the queue. This event would normally cause a JMSException to be thrown to the sender. However, because the sender in this case is another server, it cannot catch the JMSException. The message is sent to the DMQ of the sending server, Aaa.

Note This undelivered message reason code does not apply to the case where a queue is filling up and the remaining space is too small for the message. In that event, flow control is implemented and the message does not go to the DMQ.

Other Cases Where Messages Might Be Lost

The following questions provide information about cases where messages might be lost.

What are the JMSDestination and JMSExpiration values for expired messages?

When messages are enqueued in the DMQ they retain their original destination and JMSExpiration value.

Make sure that `QueueBrowsers` and `QueueReceivers` on the DMQ check the `(javax.jms.Message) m.getJMSDestination()` for the original queue. Also, checking `m.getJMSExpiration()` will always yield a time in the past.

Can the DMQ be used in a "Denial of Service Attack" to shut down the server?

If a routing user does not have permissions to write to a particular queue, messages arriving from this routing node will be dropped regardless of their `JMS_SonicMQ_preserveUndelivered` property. That is, they will not go to the DMQ.

Are there other cases where messages with a setting for JMS_SonicMQ_preserveUndelivered are lost?

Messages sent with a NON_PERSISTENT delivery mode are subject to a lower quality of service than PERSISTENT messages. The DMQ is designed to act like any other queue in SonicMQ (except where specifically noted previously). Therefore, NON_PERSISTENT messages in the queue will not be retained after either a planned or unplanned shutdown of the server.

These messages must be processed in the same server session where they occurred, otherwise they will be discarded.

If the servers stay up, but the network fails, can messages be lost?

There is one case where this can happen when messages are sent with a NON_PERSISTENT delivery mode. When routing occurs between servers, NON_PERSISTENT messages are not subject to the same level of acknowledgement as `PERSISTENT` messages. In this case, one routing node could send a NON_PERSISTENT message to another node and the network could fail. Additional messages will be blocked at the originating server pending re-establishing the connection, but a message that was indoubt might be lost if it was sent with a NON_PERSISTENT delivery mode.

Chapter 4

Failover and Load Balancing

SonicMQ implements two features which you can use singly or together to use your resources efficiently and reliably:

- Connect-time failover lets a client (or routing node) connect to any server in a list that you supply, so a connection can be made even if some of the servers in the list are not available. This is covered in the "Connect-time Failover" section.
- Load balancing lets a client (or a server acting as a client) be redirected to another server for the purpose of redistributing load. This is covered in the "Load Balancing" section.

Connect-time Failover

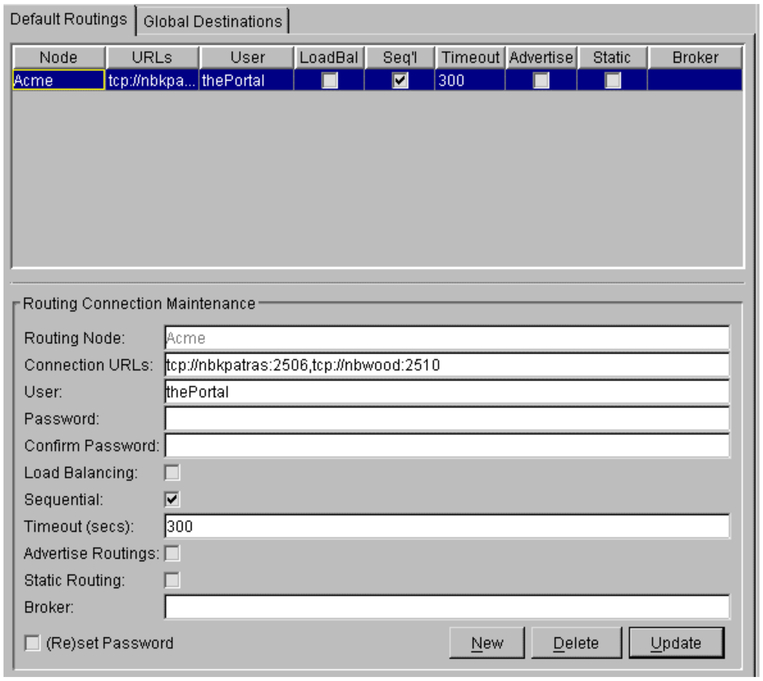
Connect-time **failover** is based on a client (or routing connection) specifying a list of servers in a cluster to which it might initially connect. If one connection attempt fails, other connections from the list will be tried until either a connection is made or a timeout condition terminates the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

You can specify a list access method, which determines which server will be tried first. This can be either **sequential** or **random**. With the sequential method, the first server in the list will be tried first. Sequential start is simplest and works well for most applications. With random order, the server first tried 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. With either sequential or random start, subsequent connection attempts will be made in the order in which the servers occur in the list.

Failover and Routing

For routing connections, one server acts as a client and the other server is usually part of a cluster.

Figure 10 shows a routing table with sequential failover selected for two connections.



Node	URLs	User	LoadBal	Seq'l	Timeout	Advertise	Static	Broker
Acme	tcp://nbkpa...	thePortal	☐	✓	300	☐	☐	

Routing Connection Maintenance

Routing Node:	Acme
Connection URLs:	tcp://nbkpatras:2506,tcp://nbwood:2510
User:	thePortal
Password:	
Confirm Password:	
Load Balancing:	☐
Sequential:	✓
Timeout (secs):	300
Advertise Routings:	☐
Static Routing:	☐
Broker:	

Figure 10. Failover and Load Balancing for a Routing Node

Defining the List of Connection URLs

All outbound routing connections from a routing node can be configured within the routing node at the server or at the configuration server for the cluster.

When you create a load-balanced connection to a client, the Connection URL list parameter lets you connect to the first available server in a list, which in most cases maps to a subset of a cluster.

The list of server Connection URLs is separated by commas. The list can be up to 4,000 characters long, and each element must be a valid URL. The following example is a valid connection URL list:

```
myserver1:2506,myserver2:2507
```

Client Access to Failover Connections

It is also possible to use failover connections from a messaging client.

This feature is available from the ConnectionFactory objects and is specific to the SonicMQ APIs. It is not JMS-standard.

The specifics on using load balancing from a client can be found in the javadoc for the `progress.message.jclient` package. The relevant classes and methods for both `TopicConnectionFactory` and `QueueConnectionFactory` are:

- `setConnectionURLs(…)` and `getConnectionURLs()`
- `setSequential(…)` and `getSequential()`

Load Balancing

Load balancing is a method of distributing connections over several servers in a cluster to avoid creating a bottleneck that might result from overloading a server. SonicMQ implements load balancing by using a round-robin algorithm. Load balancing occurs at connection time and cannot be dynamically changed without closing the connection and creating a new one.

By default, round-robin load balancing is enabled for all servers. To turn off load balancing at a server, set the `ENABLE_LOADBALANCING` property to `FALSE` in the `broker.ini` file. The client (or routing connection) must explicitly ask for load balancing as part of the connection settings.

You can reconfigure a cluster while the round-robin load balancing agent is running: the agent will include new servers for round-robin connections and stop redirecting connections to servers that have left the cluster.

If the load-balancing parameter for a connection is set to `TRUE`, load balancing is enabled for the connection. This indicates that the client (or server acting as a client) is willing to have its connection redirected to a different server. If load balancing is enabled on the server side, a client can still explicitly enable or disable load balancing for a particular connection request. The redirection happens transparently to the client.

Load Balancing and Routing

Connection balancing is performed within a cluster. Connections are bidirectional and are reused as much as possible for routing.

When you connect to a server in a cluster that has load balancing enabled and ask for a load-balanced connection, you are returned a URL that redirects you to a different server in the cluster. The redirected URL will be the default acceptor for that server.

That is, if the servers in the cluster have multiple acceptors defined (`NUM_ACCEPTORS=`*n*, where *n* > 1), only the first acceptor is used for redirecting a connection.

Figure 10 on page 60 shows a routing table with load balancing selected.

Client Access to Load-balanced Connections

You can also use load-balanced connections from a client. You should not enable load balancing if your client application always has to connect to the same physical server.

This feature is available from the ConnectionFactory objects and is specific to the SonicMQ APIs. It is not JMS-standard.

The specifics on using load balancing from a client can be found in the javadoc for the `progress.message.jclient` package. The relevant classes and methods for both `TopicConnectionFactory` and `QueueConnectionFactory` are `setLoadBalancing(…)` and `getLoadBalancing()`.

After Connecting

Once you have a load-balanced or failover connection, you can query where the connection ended up by using the `getbrokerURL()` method on the connection.

Note Failover can specify a list of servers that may or may not be part of a cluster. However, load balancing can only occur across clustered servers.

Chapter 5

Security

This chapter consists of a number of sections dealing with security topics:

- "SonicMQ Security Basics" gives a detailed overview of the way SonicMQ helps you address security concerns.
- "Client-side Security Issues" deals with HTTP tunneling, and forward and reverse proxies.
- "Signed Applets" gives details on how to use signed applets and Java plug-ins to overcome the Java sandbox security restriction.
- "Certificate-based Mutual Authentication" describes how SonicMQ supports mutual authentication for both sides of an SSL connection.
- "Password-based Encryption (PBE) Tool" describes a command line tool for encrypting the SonicMQ `broker.ini` file.
- "SSL Support" mentions SonicMQ's support for IAIK SSL and directs you to configuration information.
- "Certificate Management Tools" mentions the GUI tools for managing SSL certificates and directs you to usage information.

SonicMQ Security Basics

In a common SonicMQ configuration, one application communicates asynchronously with another across the Internet. The SonicMQ client embedded in an application communicates with another application by sending messages to a message server. One or more SonicMQ clients then consume messages from the server. All these applications are clients of a SonicMQ server.

The Need for Security

The business data encapsulated in a message might be of a highly confidential nature. A company’s continued success depends on retaining private information such as a customer’s credit card number, design specifications for an upcoming product, or the details of a sealed bid. You must also ensure the integrity of business information; that is, you must prevent an attacker from changing the content of a message.

In addition to maintaining the privacy and integrity of messages, a messaging system must be configured to prevent malicious users from compromising your computer system in some other way, such as by accessing a database or erasing files.

Security Tools

SonicMQ supplies tools that allow you to:

- Protect messages sent and delivered
- Secure the connections over which the messages travel
- Limit access to the messaging system to authorized users only
- Limit access to specific messages to authorized users only

SonicMQ also works with third-party firewall products that enable you to protect your internal network from individuals with malicious intent.

This chapter describes how you can use these security tools to protect your SonicMQ applications.

Overall Security Policy

Securing a SonicMQ application should be part of an overall corporate security strategy that protects not only the SonicMQ server, but also all applications and data that a corporation wants to shield from attack. After all, a Web site is just a public place on a corporate network allowing access to Internet users.

Long before SonicMQ is deployed on a corporate network, the administrator of the network should recognize that significant threats exist and that security solutions are needed to lower the risk of someone gaining illegal access to the corporate network. In short, the administrator must realize the need for a set of corporate security rules that dictates what is and is not allowed to happen on the corporate network.

There should be at least one person at your site whose job it is to administer security. That person is the **Security Administrator**. The Security Administrator manages SonicMQ security by using the SonicMQ Explorer, the Admin Tool, or the Management API. The administrative tool connects to a server that centrally administers message security. If you are using a cluster of servers, the configuration server handles security administration for the entire cluster.

When many people think about securing an Internet application, the first thing they consider is setting up a firewall. Establishing and implementing firewall architecture is an important and complex topic, and this document devotes a major subsection to it. Other security issues including authentication, authorization, and encryption are covered in the *SonicMQ Installation and Administration Guide*.

Corporate Security Policy

The first step in ensuring corporate security is to form a security policy. A **security policy** is 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. For example, you might not want to allow TELNET requests into your corporate Web site. In addition, you might not want corporate users to access particular Web sites available on the Internet. Both of these are examples of rules that should be defined in your security policy. After determining the policy you want to adopt, it is time to implement it.

Security Issues Covered Elsewhere

A number of security issues that you must consider are covered in the *SonicMQ Installation and Administration Guide*. These include:

- User authentication
- User authorization
- Encryption at the message level
- Quality of Protection (QoP)
- Access control lists

SSL Support

Sonic MQ supports encryption at the connection level through SSL. SonicMQ ships with BSAFE-J SSL by RSA Security to insure secure connections. Sonic MQ also supports (but does not include) IAIK (Institute for Applied Information Processing and Communications) SSL. See the *SonicMQ Installation and Administration Guide* for more information about SSL and for directions for configuring SSL.

Certificate Management Tools

SonicMQ supports a suite of Certificate Management Tools which are integrated into the Explorer administration tool. For information on the Certificate Management Tools, see the *SonicMQ Installation and Administration Guide*.

Securing the SonicMQ Data Store

To keep a SonicMQ installation secure, the data store must be secure since it contains sensitive information such as:

- Persistent messages
- User names and passwords
- Access control lists

One way to secure the data store is to limit access to it. In addition to maintaining the access control lists in a secure manner, you also need to limit access to the persistent data store through other means. If you are using an external DBMS, you should use a separate database for SonicMQ security and you should restrict access to the database to SonicMQ clients only.

Maintaining Security

After setting up your firewall and implementing your security policy, it is critical to keep the system working properly. Here are some tips for maintaining your security system:

- Perform regular system backups. If a system is penetrated, this will enable you to recover information that has been hacked or destroyed.
- Manage user accounts properly. Personnel come and go from companies, so be sure to close accounts quickly so they cannot be exploited.
- Keep hardware and software up to date. As hackers find new ways to exploit perimeter defenses, the companies who produce the defenses release new products to prevent those break-ins. If you fail to upgrade your defenses, you leave your network open to damage by hackers exploiting well-known problems.
- Monitor log files, audit trails, and alarms. These are the mechanisms that a firewall and its components use to enable a network administrator to discover potential problems. Pay attention to them. They might alert you to potential problems before any damage is done.
- Respond to attackers. If you identify the source of an attack, alert the appropriate Internet service provider.

Firewall Architecture Basics

Firewall implementations can be designed using a variety of architectures. To ensure the highest level of security you should use a screened subnet architecture to set up your firewall.

Figure 11 shows a typical screened subnet architecture using an exterior router (sometimes called an access router) and an interior router (sometimes called a choke router).

One of the most important features in this diagram is the **Demilitarized Zone (DMZ)**. Its job is to provide a medium-security zone that is accessible to the

Internet while isolating your application. If someone does get into your DMZ machine, your application data is safe on the inside network. The DMZ area is protected by the firewall, but does not expose the ports used to communicate with the inside (most secure) network to the outside (least secure) world.

An important component of the DMZ is the **bastion host**, a host machine whose address is known on the Internet. Due to its exposed position, it is fortified by removing any unnecessary applications that might compromise security.

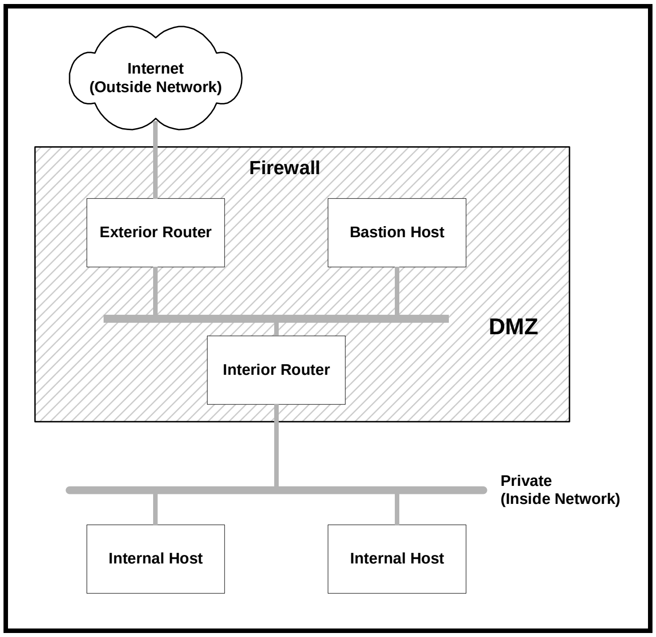


Figure 11. Screened Subnet Architecture

Many variations of this architecture exist. This diagram outlines only the most common type of screened subnet architecture. This architecture uses a DMZ to give two major advantages:

By creating a separate network as a DMZ, you can set up a configuration that uses both packet filtering and proxy server technology. This helps with the diversity of your defenses and makes your internal network very hard for outside users to reach. Many of the current firewall products use a combination of techniques so that you can customize your system to meet your security criteria.

A DMZ minimizes the impact to your internal network if the bastion host is compromised. This DMZ can contain another network, a series of routers, or even another firewall to add another level of security protection should the bastion host be compromised.

In recent years, it has become common practice to add additional DMZ machines or networks to some firewall configurations. The value of this practice is controversial. The main reason it is done is to set up a special DMZ network for particular users coming in from the outside network. For example, if you have a special partnership with a company called Acme Products, you might want to set up a special area for them to access through the Internet. The firewall can be configured to allow traffic from the Acme corporate IP address to a special DMZ network where you might allow them access to more of the inside network than users routed to the normal DMZ network. The disadvantage here is that you complicate the set of rules that the firewall has to deal with and open more holes in the firewall to be exploited. Use this variation of the screened subnet architecture with care.

The screened subnet architecture is well suited for a SonicMQ configuration because it allows you to separate application components from each other and protect them individually.

Figure 12 shows a sample browser-based application for processing order entries.

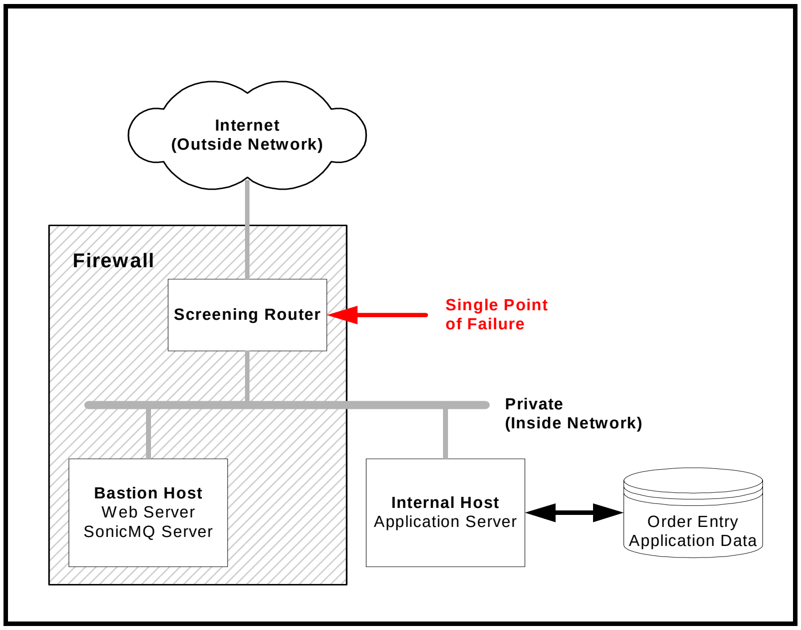


Figure 12. Screened Subnet Architecture with SonicMQ

With this screened subnet architecture, the SonicMQ server and Web server can be removed from the inside network, where the databases, application code, and sensitive private data reside. They are removed to an area of medium security where they can still bridge the gap between the end user’s Web browser looking for order status and the application code and data. However, if the network they reside on is compromised, the database for the order-entry system and the application to access it are still protected.

SonicMQ Firewall Architecture

The best solution when building a firewall is seldom a single technique. It is usually a combination of techniques implemented to solve the user requirements at a particular site.

The recommended SonicMQ firewall architecture is a variation of the screened subnet architecture. This architecture is sometimes called a three-legged architecture in that it deals with three main networks: the outside network (least secure), the inside network (most secure), and the DMZ (medium security). When designing an architecture for protecting your SonicMQ configurations, you might have to make compromises to suit the needs of your particular users.

Recognizing that fact, Progress recommends an architecture that is as robust as possible, but that still provides the best firewall security strategy. This robustness is built into the firewall architecture by using network hubs and network routers placed at various locations. Adding network hubs allows you to reconfigure your firewall system as the need arises and limits the number of changes that will be required when this time comes. For example, site security policies might require you to add another machine to the DMZ network. If you have a network hub in place on your DMZ network, you can simply add the other machine to the hub, and no other rewiring or configuration is necessary. Adding network routers involves adding extra security measures as well. This is primarily because most routers include software that allows you to add access rules and to notify a system administrator about potential attacks.

This architecture also gives you a depth to your firewall; that is, it has several points where you can make security checks so that a single failure will not leave your system open to malicious attacks.

Machines that reside in the DMZ and host the Web server and the SonicMQ servers should be battle-hardened. That is, you should disable FTP access and remove any unnecessary software that presents security risks.

You should implement this architecture with a diversity of defenses in mind. It is not only important to use a number of different systems for your firewall defense, but also to use hardware and software from different vendors. The reason for doing this is simple: There might be a bug in a particular vendor’s hardware or software that can compromise your system. Having systems from different vendors reduces the risk of your whole system being open to someone who uses that bug to compromise your system. For example, use a router from one vendor on your line to the Internet (your external router), but use another router from another vendor on the line to your inside network (your internal router). If someone gains access from the outside by exploiting a router bug in the external router, they will not be able to exploit the same bug to gain access to the inside LAN, which has a router by a different vendor.

Figure 13 shows a recommended firewall architecture for the server side of a typical SonicMQ server configuration.

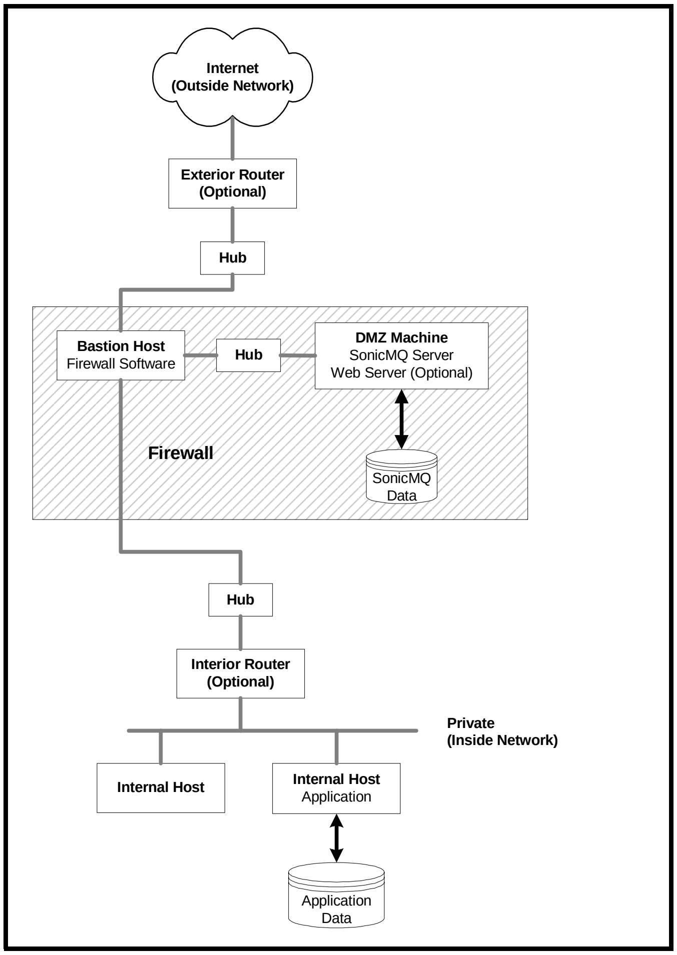


Figure 13. Recommended Architecture: Variation I

With this model, the firewall software is performing the role of both an interior router and an exterior router.

Depending on your needs, you might want to have the SonicMQ server placed in your private inside network where you could install it on a machine non-battle-hardened machine. You can do this using an off-the-shelf reverse proxy server. See the *SonicMQ Release Notes* for the latest information about supported reverse proxy servers for SonicMQ.

If you choose to vary the screened subnet architecture, you could place the reverse proxy server in the DMZ and the SonicMQ server in the inside network, as shown in Figure 14.

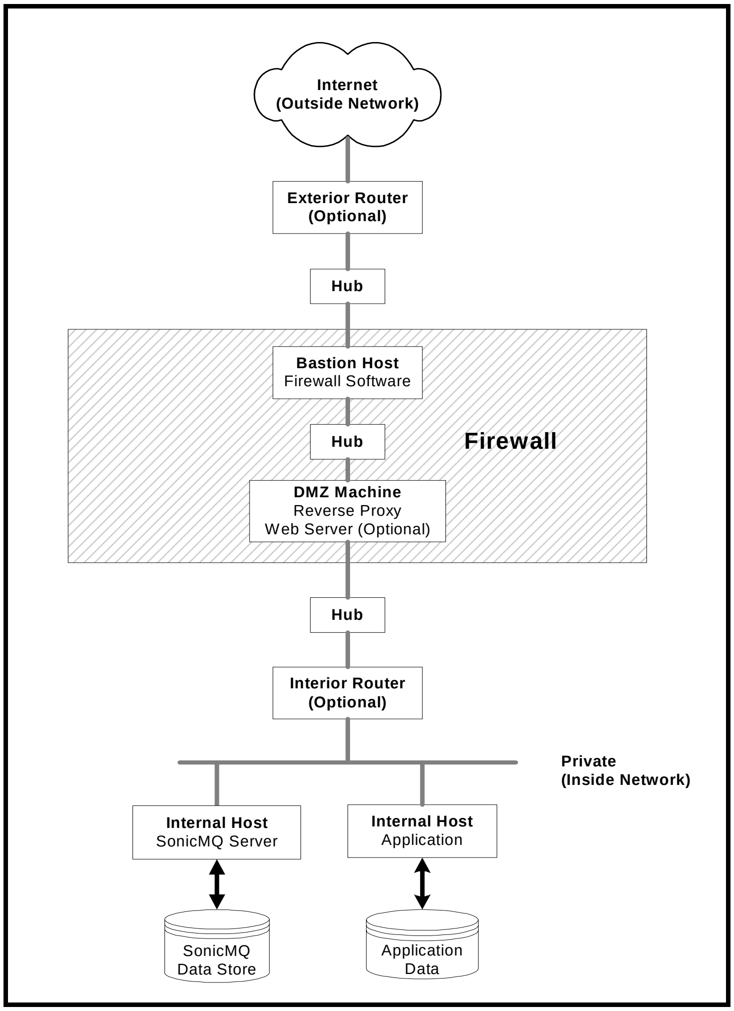


Figure 14. Recommended Architecture: Variation II

Advantages of the Screened Subnet Architecture

The screened subnet architecture is more secure than router-based solutions because today's firewall software products provide much better protection than the software typically used on a standard router. In the recommended SonicMQ firewall architecture, a host-based PC or UNIX workstation hosts the firewall software. This allows Web sites that cannot afford expensive hardware to implement this architecture effectively. However, you might want to add optional routers to the recommended architecture, depending on the complexity of your internal network or your budget. This provides an outstanding way to add extra levels of defense.

In the first recommended variation, the Web server and the SonicMQ server reside in the DMZ. Internet users can access these portions of the SonicMQ application, but cannot directly access the main portion of your SonicMQ application. The firewall is configured so that only the SonicMQ servers themselves are allowed to talk with the inside network.

The remainder of the application resides on the inside network. In the Figure 13, the components all reside on one host machine.

In the second recommended variation, the SonicMQ server resides in the private inside network, and its address remains unknown to Internet users who communicate with it through the reverse proxy server. In effect, this adds an extra layer of security to your SonicMQ application.

Setting the Firewall Rules for a SonicMQ Application

This section explains how to use the firewall to protect the DMZ from the outside network and using the DMZ to protect the inside network. You must develop a set of rules to give to the firewall to set up this protection. Each firewall system (hardware or software) will have different ways to set up these rules. Check the documentation for the product you are using.

To set up your firewall, start the firewall software with no rules defined. Then systematically add rules and test each rule as it is added. When you initialize the firewall and there are no rules defined, you should not be able access any host machine from the outside network.

Most firewall products require you to configure a Domain Name Service (DNS), which runs on virtually all host machines. It translates the IP address associated with a machine to a logical name. You should use DNS names to avoid having to repeatedly supply 10-digit IP addresses.

Adding and Testing Your SonicMQ-specific Rules

Once you have configured DNS (if required) you are ready to define your rules. There are a variety of ways to test generic rules for applications including HTTP, FTP, and TELNET.

The first rule you set up should allow access from the outside network to the Web server on the DMZ. Once you have set up this rule, test it by starting a client on the outside network and have it connect to the DMZ machine. This can be done by pinging the machine, or if ping is not supported, by telneting to the port. If the connection is successful, proceed to the next rule. If not, recheck your rules in the firewall rules database.

Begin adding the rules and test your configuration after adding each one. Depending on your firewall, you might need to save and reconfigure after adding each rule. If you do not, the rule might not be applied and the result could be confusing. Once you have supplied these rules to your firewall software, you can test your SonicMQ configuration components together.

All firewall products keep extensive log files. If you make a mistake, use the log files to tell you which components are trying to access restricted ports. Close any ports you might have opened when you defined your rules. If you inadvertently leave the ports open, these holes might be exploited for attacks.

Client-side Security Issues

Client-side security involves the following topics:

- HTTP tunneling
- Use of forward proxies
- Use of reverse proxies

HTTP Overview

SonicMQ insulates the details of the protocol layer from the application developer. Which protocol layer is to be used on the client-side is determined entirely by how you specify the message server's URL to the client application.

The Hypertext Transfer Protocol (HTTP), like the TCP and SSL protocols, is always available to the client application. From the developer's perspective all three protocols behave the same way. Synchronous and asynchronous communications are both available regardless of the protocol choice, and the application does not require special coding to accommodate the protocol.

By server design or by company security policy, proxy servers and firewalls frequently only allow HTTP-based traffic to pass through.

You can establish a direct connection between client and server using HTTP Tunneling as the protocol, as shown in Figure 15. However, because the HTTP tunneling protocol is significantly slower than TCP or SSL, this option is only recommended when TCP and SSL protocols are not available.

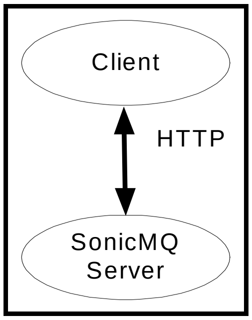


Figure 15. Direct HTTP Connection

To deploy on the Internet, you usually use HTTP. In Figure 16, the proxy server and firewall are optional components. The diagram shows that if the SonicMQ server is going to directly process messages received from the clients over the Internet, it must be deployed as if it were a Web server. It must reside on a system in your demilitarized zone (DMZ), and not on your intranet.

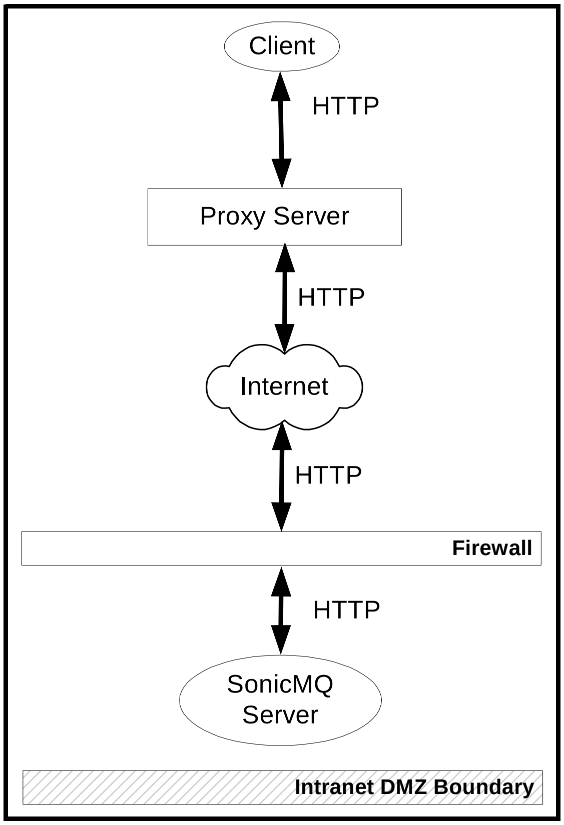


Figure 16. Internet Deployment with Proxy Server and Firewall

The requirement that a messaging server reside in the DMZ can be removed if you place a reverse proxy server in your DMZ and use it to re-direct data traffic to a server running on your intranet, as shown in Figure 17. Some Web servers can be configured to function as a reverse proxy server as well as a Web server.

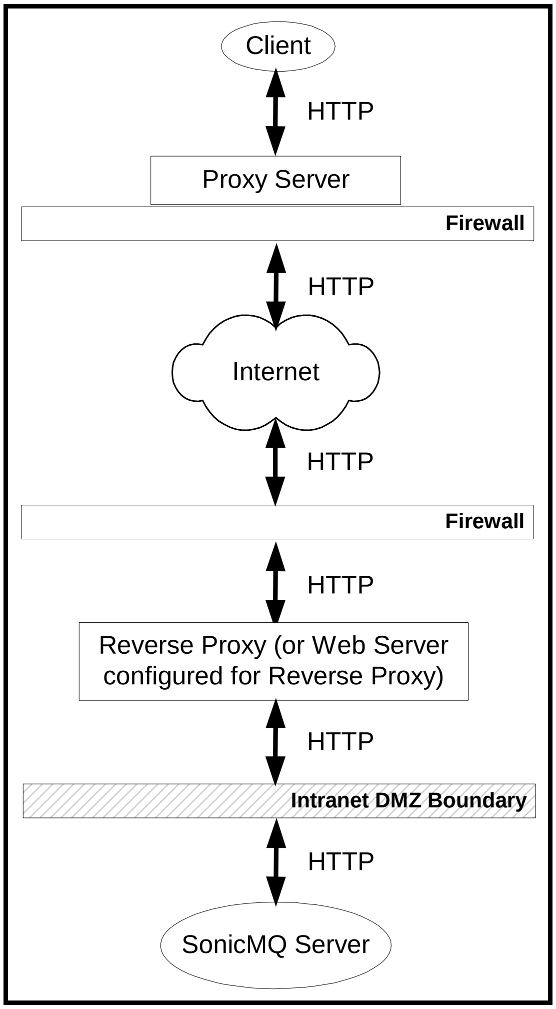


Figure 17. Internet Deployment with Reverse Proxy Server

Understanding HTTP Tunneling in SonicMQ

HTTP Tunneling supports both synchronous and asynchronous communications. The HTTP protocol is not inherently an asynchronous communication protocol, but SonicMQ makes it function as one. This is accomplished by creating multiple physical connections to the server from the client. If HTTP 1.1 Persistent Connection is available between the client and the server, SonicMQ establishes minimum of three connections:

- The first physical connection is used to initiate the JMS Connection.
- The second physical connection is used for sending data to the server.
- The third physical connection allows the server to send data back to the client.

HTTP originally allowed only one request per physical TCP connection. However, establishing a TCP connection is fairly expensive, so some implementers of HTTP/1.0 added the Keep-Alive connection header value to keep a connection open after a request was completed and to allow further requests to be made over that connection. Unfortunately, the HTTP/1.0 Keep-Alive connection header is not implemented in all proxy servers claiming HTTP/1.0 compliance. The HTTP/1.1 specification defines persistent connections and makes them the default.

The processing of the first connect request is used to determine which level of HTTP protocol support is available. The optimum situation is when HTTP/1.1 Persistent Connections are available, so the cost of creating the physical TCP connection is paid only one time. If HTTP/1.1 Persistent Connections are not available, the server looks for HTTP/1.0 Keep-Alive Connections. SonicMQ reuses connections as much as possible, minimizing the cost of creating the physical connections. The lowest, and slowest, level is when HTTP/1.0 without Keep-Alive is the only level available, which might well be the case if a client-side proxy server is between the client and the server. This level is slowest because a physical connection must be created for each request posted from the client to the server.

HTTP Tunneling

This section presents a procedure for using HTTP tunneling.

➤ **To use HTTP tunneling:**

1. Use the SonicMQ `webclient.jar` file to find the JMS client classes instead of using `client.jar`. The `webclient.jar` file includes the HTTPClient package.

2. Direct the client to use the HTTP Tunneling network protocol by beginning the URL string for the server with `http://` as in the following line of code:

   ```
   TopicConnection myconn = new
   TopicConnection(“http://myserverhost:80”, appid, usrname, passwd);
   ```

3. Configure the server to receive HTTP connections. Edit the following lines in the `broker.ini` file:

   ```
   ; Set protocol
   DEFAULT_SOCKET_TYPE=tcp
   ;DEFAULT_SOCKET_TYPE=http
   ;DEFAULT_SOCKET_TYPE=ssl
   ```

   by commenting out the second line and uncommenting the third line as follows:

   ```
   ; Set protocol
   ;DEFAULT_SOCKET_TYPE=tcp
   DEFAULT_SOCKET_TYPE=http
   ;DEFAULT_SOCKET_TYPE=ssl
   ```

4. Save the `broker.ini` file.

   When you restart the server it will display a message like:

   ```
   SonicMQ Broker started, now accepting http connections on port
   2506...
   ```

Using a Client-side Forward Proxy

A client-side forward proxy (proxy server) is a third-party server which lies between one or more SonicMQ clients (or servers acting as clients) and a firewall. SonicMQ supports the standard SSL proxy. See the *SonicMQ Release Notes* for the latest information about supported forward proxies for SonicMQ. To obtain the proxy server’s host and port information, the HTTPClient package:

1. Reads the case-sensitive system properties `http.proxyHost` and `http.proxyPort` from the JVM.
2. Automatically configures itself to use the proxy server to make the connections.
3. Once the properties are set, the HTTP connections are made through that proxy server.

The properties `http.proxyHost` and `http.proxyPort` can be read in three ways:

- For applications, you can set these properties from the command line:

  ```
  -Dhttp.proxyHost=hostname –Dhttp.proxyPort=80
  ```

- The properties can be set programmatically as in the following example:

  ```
  Properties props = System.getProperties();
  props.put("http.proxyHost", proxyhost);
  props.put("http.proxyPort", proxyport);
  ```

- In an applet scenario, the browser automatically sets these properties.

Because the class that uses the properties reads them in its static initializer, they must also be set before any connection is attempted and cannot be changed later.

When run from an applet in a browser, the SecurityException message will appear in the Java Console every time the Applet starts. This exception is caught inside the initializer, but the browser's AppletSecurityManager prints the message before throwing the exception.

Applets using HTTP must be usually be signed. See “Signed Applets” on page 84 for details.

Using a Server-side Reverse Proxy

Note See the *SonicMQ Release Notes* for the latest information about supported proxy servers for SonicMQ.

If the Universal Resource Identifier (URI) for a resource request contains an `/SC` identifier:

```
http://hostname:port/SC/...
```

a reverse proxy recognizes the request as a SonicMQ HTTP request and maps and forwards the request to a SonicMQ server.

For example, you could use these lines for an Apache configuration:

```
ProxyPass         /SC http://serverhost:2506/SC
ProxyPassReverse  /SC http://serverhost:2506/SC
```

Note Off-the-self reverse proxies may have scalability limitations in the number of clients that can be supported.

Important If you use a reverse proxy server, you will not be able to use some SonicMQ features, such as SSL or load balancing. This restriction does not apply to client side forward proxies where the server is in the DMZ.

Using an ActiveX Client with HTTP Tunneling

Creating HTTP tunneling connections from an ActiveX client is essentially the same as creating connections from a Java client, with the following caveat:

The Javasoft plug-in does not currently support setting command-line parameters like the `-Dhttp.proxyHost` . There is also no way to set them programmatically from the ActiveX container.

You can use the SonicMQ ActiveX Control's `setBrokerURL( )` method to specify *`proxyHost`* and *`proxyPort`* within in the server's URL string using the following format:

```
http://serverHost:serverPort:proxyHost:proxyPort
```

If you are not using a proxy server, then you do not need the *`proxyHost`* and *`proxyPort`* strings. You can use the string:

```
http://serverHost[:serverPort]
```

If you omit *`serverPort`*, the default value of 2506 is used. However, you cannot omit *`serverPort`* if you use *`proxyHost`* and *`proxyPort`*.

Signed Applets

If you want to use applets, you are faced with the Java sandbox security restriction. If this restriction is not lifted, use of applets is limited to the simplest kind of deployment where the Web server and the message server are on the same machine and there is no proxy server between the client and the server machine.

Since you typically have no control over whether the client uses a proxy server and since you often want the Web server and the message server to be on different machines, you need to get around the Java sandbox security restriction. To do this, you must sign your applets. There are two main ways to do this:

- Browser-specific tools
- Java plug-ins

Browser-specific Tools

Applet signing is supported by Netscape® Communicator and Microsoft Internet Explorer.

➤ **To sign applets with Netscape or Microsoft browsers:**

1. Create an installable signed JAR file containing all files required by the applet.
2. Distribute the installable JAR file from the server to the user's computer.
3. Create a trigger script which determines which files from the signed JAR file actually need to be downloaded, and which are already present. (Optional, Netscape only)

Each of these steps is complex and vendor-dependent. For instructions, go to the Netscape or Microsoft Web pages.

Java Plug-ins

You can overcome the problem of the browser-dependence in the creation of signed applets by using a Java plug-in. To sign applets using a Java plug in, you must download the appropriate applet-signing tool:

- For JDK 1.1.x, download javakey
- For JDK 1.2 and JDK 1.3 download keytool

The JDKs are available as free downloads from the Sun Microsystem Web page. You will also find instructions for using the Java plug-ins on the Sun Microsystem Web page.

Certificate-based Mutual Authentication

Certificate-based mutual authentication is supported with Secure Socket Layer (SSL) for server-to-server communication and for server-to-client communication.

You can import the certificate identity, which is used as a username, directly into the user database using the Explorer. You can use the certificate identity as the routing username for access control to remote queues in remote routing nodes. To use mutual authentication you specify the special routing user name `AUTHENTICATED`. See "Connection Security" on page 121 for more information.

See Chapter 2, "Multi-node Architecture" for more about remote routing nodes.

Password-based Encryption (PBE) Tool

The **PBETool** is a command-line tool that you can use to DES-encrypt and DES-decrypt a `broker.ini` file.

When SonicMQ servers are deployed in the DMZ, the `broker.ini` file is also commonly placed in the DMZ. You might be concerned about the vulnerability of sensitive information in that file. The **`BROKER_PASSWORD`** and **`SSL_PRIVATE_KEY_PASSWORD`** are two examples of such sensitive information.

Note The `broker.ini` file need not reside on the same system as the SonicMQ server. The `broker.ini` file is opened using a `java.io.FileInputStream`, and therefore `broker.ini` can be moved to a system on the intranet outside of the DMZ which is available to the system inside the DMZ.

The SonicMQ server and the SonicMQ dbtool can read both the clear-text and the encrypted versions of the `broker.ini` file. Figure 18 shows the relationships between the components involved.

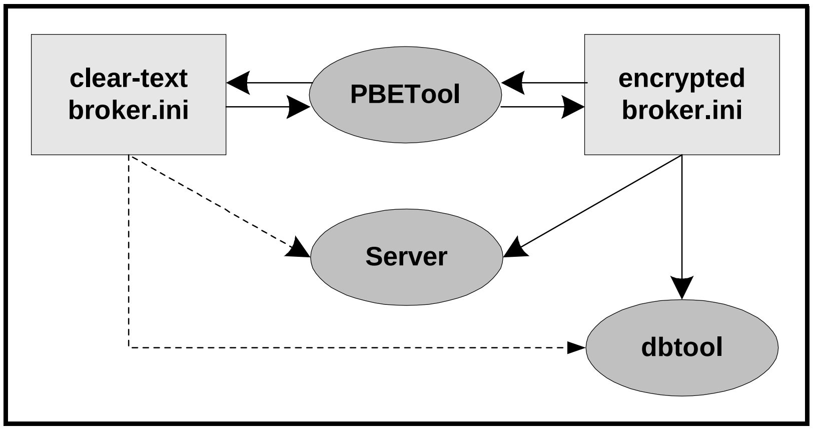


Figure 18. Password-based Encryption Architecture

The encrypted version of the file is base64 encoded. **base64 encoding** is a method for encoding binary files so that they can be transferred easily. For example, you can send base64-encoded binary files in the body of an e-mail message.

SonicMQ supplies script files to simplify the use of PBETool. The PBETool scripts set the environment for the tool and then invoke the PBETool with an input file, output file, and password which you specify.

For encryption on Windows use the command:

```
pbetool /m encrypt /c broker.ini /e encrypted_file /p password [/x]
```

For encryption on UNIX or Linux use the command:

```
pbetool.sh -m encrypt -c broker.ini -e encrypted_file -p password [-x]
```

For decryption on Windows use the command:

```
pbetool /m decrypt /c broker.ini /e encrypted_file /p password [/x]
```

For decryption on UNIX or Linux use the command:

```
pbetool.sh -m decrypt -c broker.ini -e encrypted_file -p password [-x]
```

The PBETool can only decrypt the file using the clear-text version of the password. This requirement ensures that if the encrypted password is compromised, it cannot be used to decrypt the file.

Table 6 describes the parameters accepted by the **`pbetool`** command for UNIX or Linux systems. Replace the initial dash (**`-`**) by a slash (**`/`**) for Windows systems.

Table 6. PBETool Parameters

Parameter	***Required?***	***Description***
`-m` *`mode`*	Yes	Indicates the mode that the tool will run in. Valid values for this option are `encrypt` and `decrypt`, which are case-insensitive.
`-c` *`clear-text-file`*	Yes	Specifies the name of the file that contains the clear-text data. This is the *`input-source-file`* for an encryption, or the *`output-destination-file`* for a decryption.
`-e` *`encrypted-data-file`*	Yes	Specifies the name of the file that contains the encrypted data. This is the *`input-source-file`* for a decryption or the *`output-destination-file`* for an encryption.
`-p` *`password`*	Yes	Specifies the password used to create the encryption key to encrypt or decrypt the file.
`-x`	No	Specifies that an encrypted version of the specified password will be written to the JVM standard output. This encrypted password might be used by the server to decrypt the file. If you must place the password in an insecure location, you should use the encrypted password.
`-h`	No	Displays the list of command-line options for PBETool.

Encryption

When you invoke the PBETool you specify a clear-text version of `broker.ini` as *`clear-text-file`*. See Table 6, "PBETool Parameters." The contents of this file are then read into a byte-array in memory.

An encryption key is derived from the password you provide, and the byte-array is DES-encrypted using that key.

PBETool uses a **Message Authentication Code (MAC)** to check the integrity of encrypted `broker.ini` file, based on a secret key. The MAC is produced using the cryptographic hash function MD5 or SHA1. The MAC of the original *`clear-text-file`* is generated and is embedded in the encrypted data. When decrypting the file, the embedded digest is compared against the digest computed from the decrypted data.

The `sun.misc.BASE64Encoder` class base64 encodes the entire encrypted output, including the file size. The result is written to the file you specify by *`encrypted-data-file`*.

To enhance security SonicMQ does not require you to place passwords inside script files. However, if you start the server automatically using a Windows service the password is placed in the system registry. If you use the UNIX or Linux `cron` command, the password is placed in `crontab` files. In either case, the password is placed in an insecure location.

If you must place the password in an insecure location, you should specify (using the `/x` or `-x` switch) that PBETool generate an encrypted version of the password when encrypting the `broker.ini` file. PBETool writes this encrypted password to standard output, allowing you to put the encrypted password into the insecure location. That way, you avoid storing the clear-text password in a file. If you need to read the encrypted `broker.ini` file, you must supply the clear-text password.

Warning If you encrypt the password you will need the encrypted version to set up a Windows service or `cron` command. If you lose an encrypted password, there is no way to regenerate it. Your only option is to decrypt the file with the clear-text version of the password and encrypt the configuration file again.

Decryption

The entire contents of *`encrypted-data-file`* is read into a byte array in memory. This uses the `sun.misc.BASE64Decoder` class to first decode the binary data.

An encryption key is derived from the password provided.

The length of the original file is first extracted from the data, and then the binary data is decrypted using the key.

To verify that the password provided is correct and that the decrypted data is accurate, the embedded MAC digest is compared against the digest computed from the decrypted data.

The decrypted result is written to the *`clear-text-file`*.

Using the Encrypted broker.ini File

To start a server with an encrypted server initialization file as a Windows Service, use the parameter `-inipwd=`*`clear-text-passwd`* in the SonicServiceSetup parameters. Alternatively, you can use an encrypted password by using the parameter `-encpwd=`*`encrypted-passwd`*.

To start a server with an encrypted server initialization file using the UNIX or Linux `cron` facility, you can use the clear-text password with the parameter `-p` *`password`*. Alternatively, you can use an encrypted password with the parameter `-x` *`enc-password`*.

The `startbr` and `dbtool` scripts can read the encrypted or unencrypted version of the `broker.ini` file. For these tools to read the file, they must be provided a password at startup. The password should be unrelated to any passwords contained inside the encrypted `broker.ini` file.

By default, the `startbr` and `dbtool` tools assume that the server initialization file is located in the current directory and is named `broker.ini`. When using an encrypted or renamed server initialization file, use the additional parameters described in Table 7.

Table 7. New Parameters for dbtool and startbr

Parameter	***Description***

`/f` *`path_to_INI_file`* `(Windows)`
`-f` *`path_to_INI_file`* `(UNIX/Linux)`
	Tells dbtool where to find the server initialization (INI) file. If no `-p` option is specified with this option, the server initialization file is assumed to be in clear text. If a `-p` option is specified, the server initialization file is assumed to be encrypted.

`/p` *`clear-text-password`* `(Windows)`
`-p` *`clear-text-password`* `(UNIX/Linux)`
	Specifies the password that will be used to decrypt an encrypted server initialization file.

`/x` *`encrypted-password`* `(Windows)`
`-x` *`encrypted-password`* `(UNIX/Linux)`
	Specifies the encrypted password that will be used to decrypt an encrypted server initialization file. (This does not apply to `dbtool`.)

Important When using `dbtool`, the parameters `-f` *`path_to_INI_file`* and `-p` *`clear-text-password`* or their Windows equivalents **must** be the initial parameters in the command. For example:

```
dbtool /f encrypted_file /p password /c basic (Windows)
dbtool.sh -f encrypted_file -p password -c basic (UNIX or Linux)
```

Chapter 6

Designing Messaging Models

This chapter shows some concepts of how you might deploy SonicMQ. Clarifying some of the topologies can help you take advantage of SonicMQ's features in your application—whether it be basic messaging, a supply chain, Enterprise Application Integration (EAI), or Portal with Trading Partners.

The flow of data during its time in a messaging system has several functions:

- **Business Application Services** — The fundamental messaging activity is its integration with the applications that measure and record business and real world activities.
- **Validation** — The message and its data can be verified to ensure that it is well-formatted and contains valid values. This could be done as soon as the message is composed, or when the message is received. The former adds overhead to message packaging while the latter adds a function at a point where messages that are not acceptable cannot be corrected.
- **Transformation** — A message might not be easily consumed by a single target application. The message might have to change its type from an XML message to a text message, or the message body may have to be split up. For example, a message order for a bundled product—a computer with cable and printer—could spawn multiple messages to other channels.
- **Routing** — The ultimate destination of a message might be unknown when a message is initiated. If there is any way a message can look up some information it can save steps in reaching its goal.

These functions, and the point at which they are applied, have a significant impact on the overall performance of a messaging system.

This chapter presents information about client applications then discussions on the following topologies:

- "Topologies" on page 98 describes a chain topology, consisting of a linear sequence of routing nodes.
- "Hub and Spoke" on page 100 describes a topology of a hub having any number of spokes connected to the hub.
- "Central Hub" on page 101 describes a topology where a node can connect to another node, thus creating a spoke connection to the central hub.
- "Peer-to-peer" on page 104 describes a topology where a web nodes can communicate directly with each other.

Client Functions

What you can cause to happen through client applications combines with the features of the message server architectures to determine just how you can set up a deployment.

Agent Applications

Systems that are linked to record-keeping systems are normally the starting point and endpoint in a message lifecycle. Real-time devices and accounting document lifecycles create messages that are moved into the messaging stream by agent applications. Correspondingly, receivers gather appropriate messages to funnel into their application. Figure 19 shows a business application where **A** produces and consumes messages at destinations on message server **1**.

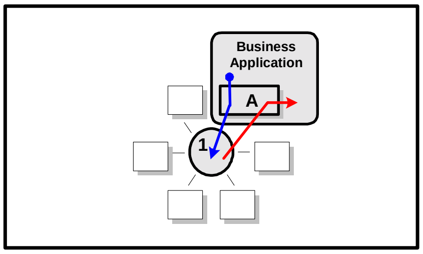


Figure 19. Agent Application

Transformation Applications

A transformation application watches for messages so that embedded business logic can transform the message 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.

As Figure 20 illustrates, an application receives a message from a message server, probably by first qualifying the messages that it can service. Then a properties or XML configuration file might provide modifiable business rules that provide the methods to unpack the message payload, determine how to route the content elements, and then send the transformed message set.

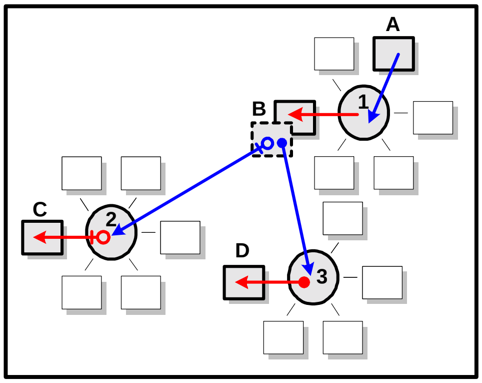


Figure 20. Transformation Application

In Figure 20, 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 meta-data—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.

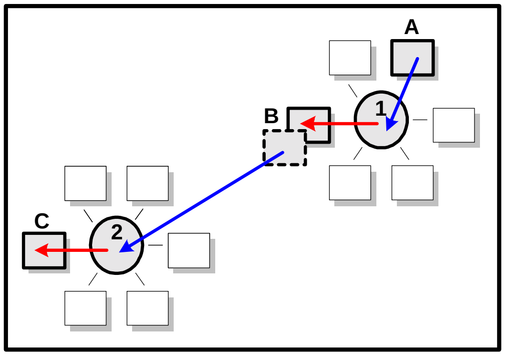


Figure 21. Routing Application

In Figure 21, 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 22, 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.

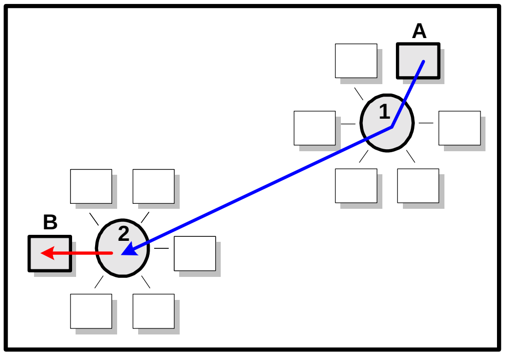


Figure 22. Dynamic Routing's Store-and-forward mechanism

The syntax of the queue name is **`<remote_node>::<remote_queue>`**. When application **A** wants to send a message to a queue on node **2** it might not be authorized to connect directly to node **2**. But the node where **A** is authorized to connect, node **1**, might have an entry in its routing table for *`Queuename`* on node **2**. If so, application **A** can send a message **node 2::Queuename**. The message would hop through node **1** and be accessible by receivers of *`Queuename`* on node **2** such as application **B**.

Node to node connections can be focused around one central hub where only a limited set of controlled applications, such as **M**, can connect directly to the central hub as shown in Figure 27.

Topologies

Chain

In a chain topology, a series of nodes, each containing a SonicMQ message server, are connected together. You can create applications for each of the servers to enable the servers to send received messages from one hub to another hub. This is essentially a linear chain of routing nodes. Figure 23 shows an example of this configuration.

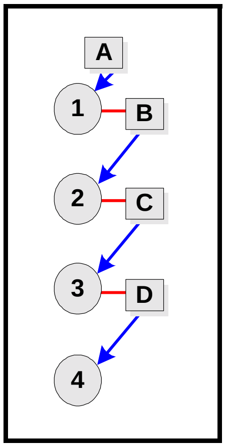


Figure 23. Chain Topology

In this example, routing node **A** can send a message to routing node **1**. The routing application **B** can receive the message and then forward it to node **2.**

When similar applications exist as receivers on a series of servers, a chain structure emerges. The disadvantage to any chain is a weak link. Here, if one application forwarding is lost, the chain might end at the last connection.

The chain topology is sensitive to any client or server going offline. However, when adequate steps are taken to persist messages and provide load balanced, reliable connections, the chain topology.

Much of the inherent risk in a simple chain topology is handled by SonicMQ's Dynamic Routing Architecture (DRA), as shown in Figure 24:

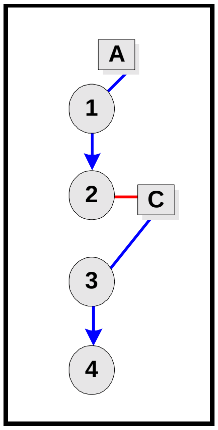


Figure 24. Enhanced Chain Topology Through Dynamic Routing

In the enhanced chain topology, a single routing application carried a message across four servers. The Dynamic Routing Architecture adds leverage to transformations. In Figure 25, the routing application **C** traverses 6 servers.

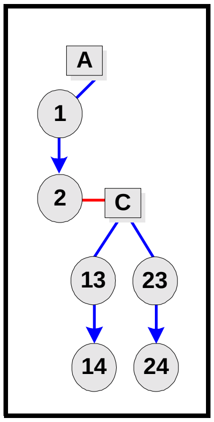


Figure 25. Chain Transformation Topology with Dynamic Routing

Hub and Spoke

The basic client-server model, the hub-and-spoke model, features a central hub having any number of spokes connected to the hub. In this topology, the clients can communicate only with the hub; the clients cannot communicate directly with each other. Figure 26 shows a hub-and-spoke topology with SonicMQ clients **A** through **F**, each located at the end of a spoke. Each client has a connection to the hub.

The hub is presented as a SonicMQ node. A node can be a message server or a cluster of message servers having shared security.

In this example, SonicMQ client **A** can communicate with client **E** by sending a message to the hub. The message server at the hub then processes the message, making it available to the intended recipient, client **E**.

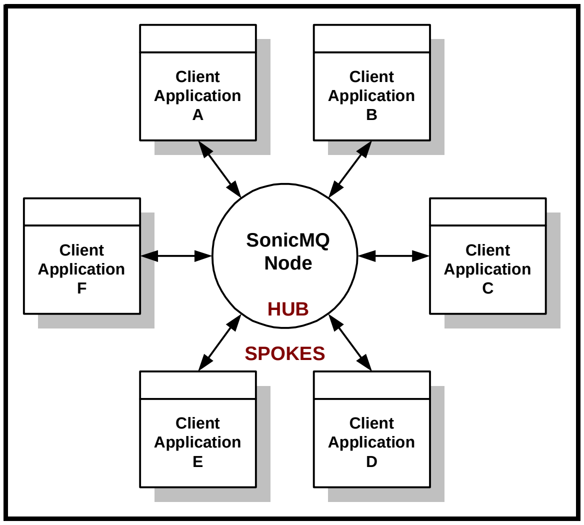


Figure 26. Hub and Spoke Topology

In practical terms, the message server never sends a message to a recipient. But if client **A** and client **E** agree that the queue (or topic) named, say, **AandE**, is “their” channel, they can set security to allow only their clients access. This creates an indirect, dedicated delivery destination. The only significant issue in this case is who has administrative privileges over access control lists.

Central Hub

When a node can connect to another node, the first node creates a spoke connection to the central node, thus creating a central hub topology. This topology is feasible because of SonicMQ's Dynamic Routing Architecture (DRA) uses relationships and registered routing routes so that an application can be connected to a node and send a message directly to a global queue on a remote node. See Chapter 2, "Multi-node Architecture," for a complete description of DRA.

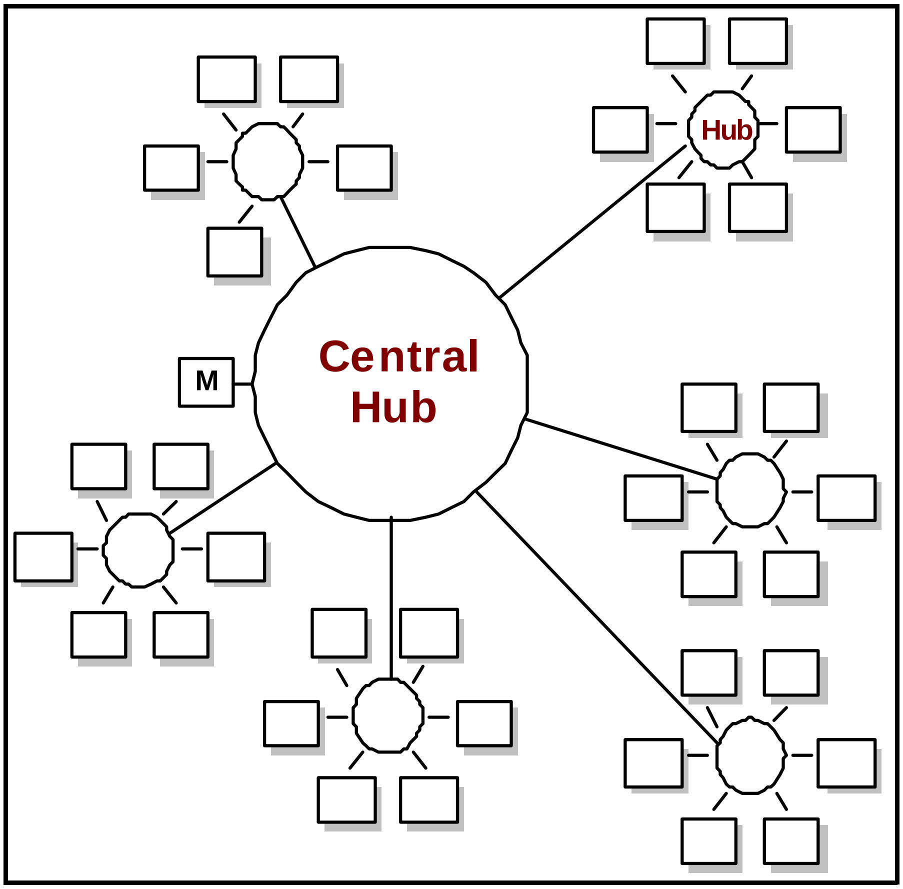


Figure 27. Central Hub Topology

The central hub model is the essence of the marketplace model as shown in Figure 28.

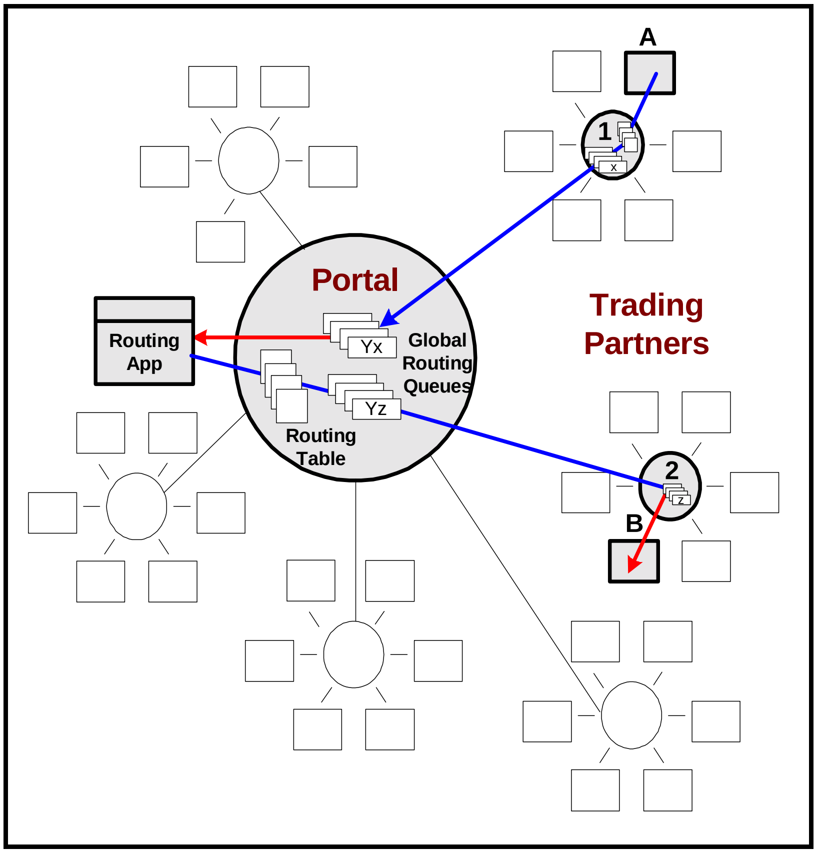


Figure 28. Central Hub with Application Control (Marketplace)

In the marketplace diagram, client application **A** connects to local message server **1** that has a routing queue that can store a message where the destination **`Portal::X`**—the **`<remote_node>::<remote_queue>`**—is listed in the routing table and then forward it to **`Portal`** message server's global routing queue **`x`**.

There, a **Routing App** receives the message on behalf of the marketplace and examines it to determine where it should be rerouted. The hints for the next destination are business rules that might be:

- User-defined properties such `AIA_Phase = Finishes` or `SIC_code = 2345.` Properties are accessible to message selectors so that routing applications only receive known message categories.
- Manifest data stored in a message body such as **XPath** info in an XML header. However, routing applications cannot ensure the integrity of a message body, especially if it is decrypted and re-encrypted.

The **Routing App** sends the cloned message to an appropriate message server where the clients are all in that market, in this case, message server **2**.

The Portal's routing table routes the message to the destination **`2::y`** as listed in the routing table. The message is stored on the portal and, when connection is available, it is forwarded to message server **2**'s queue **y**.

Assuming client application **B** was receiving with an inclusive message selector on the **y** queue, **B** takes the message as the final receiver.

The message taken by **B** could be directed to an application where it will be assimilated and transformed such as an open order becoming an invoice. Or the message could continue to be routed through other portals.

Peer-to-peer

While the structure of portals and trading partners might seem rigid, nothing prevents the trading partners from establishing direct connections, as shown in Figure 29. In the figure, **TP1** is a trading partner on Portal. **TP1** finds it in its business interest to establish direct connections to some of its other trading partners such as **TP2**, **TP3**, and **TP4**.

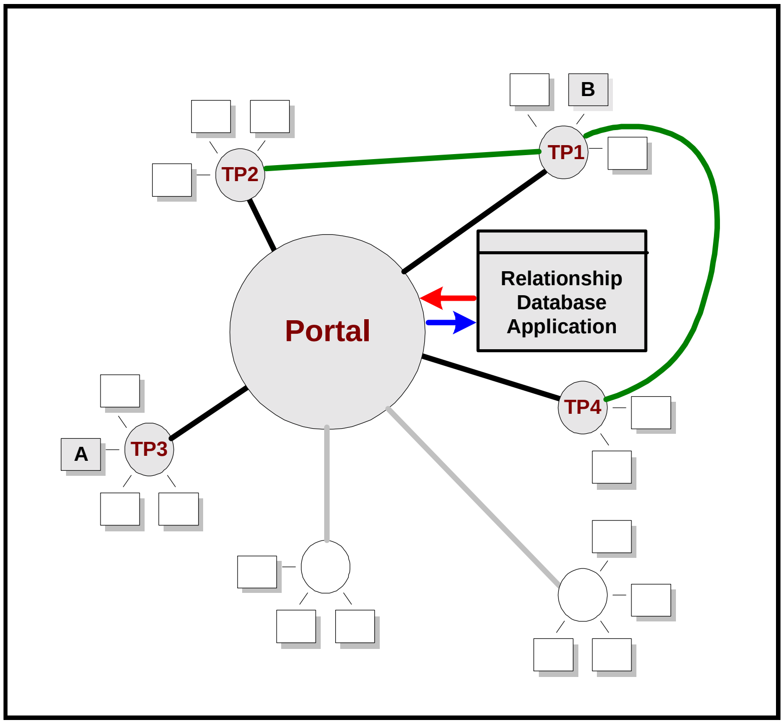


Figure 29. Peer-to-Peer with a Central Hub

In this example, routing node **A** seeks to establish a connection with routing node **B**. **A** first connects to the hub where some mechanism like a “Relationship Database” provides the meta-data necessary to establish this connection to **B**. **A** then connects directly with **B** using the information it obtained from the hub. All of the routing nodes are peers in this example, and each routing node can connect to another by first obtaining the connection information from the hub, then directly connecting to the other routing node.

Store and Forward

Your application's architecture can take advantage of routing to maintain:

- **Lower expenses** — As shown in the example Figure 30, connection from New York to Paris and other locations might be an expensive measured line. The traffic can be batched until the server is ready to send messages from New York and the server in Paris is ready to receive messages.
- **Higher efficiency** — The New York server can store messages locally and not maintain a remote connection.
- **Connection independence** — The client maintains a connection to the local server in New York, and does not care whether the connection to Paris is established. The message is sent when the connection is established.

Store-and-forward routing allows the message server to store messages until a number, size, or elapsed time indicates to a monitoring application (such as management functions or a queue browser) that a connection should be established and the messages transferred. See Chapter 2, "Multi-node Architecture," for more information about store-and-forward routing.

Figure 30 illustrates an example of peer-to-peer routing using store and forward routing.

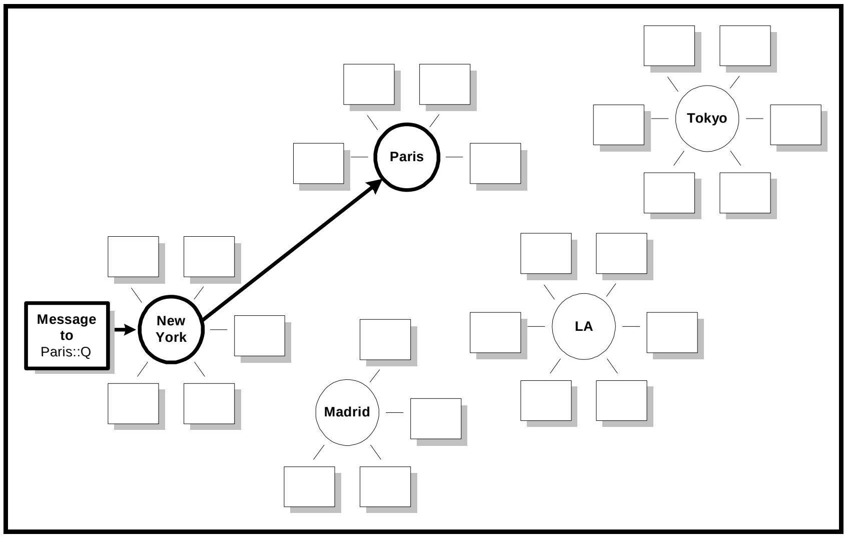


Figure 30. Peer-to-peer Topology for Store-and-forward Routing

A client on a **New York** message server is sending a message to the **Paris::Q** destination. The **New York** server might offer immediate connection to high priority messages and retain messages for other remote servers until it is triggered to connect to the remote server and send the messages to the remote server’s queue, **Q** on the **Paris** message server in this example.

Part II

Implementing Your Deployment

This part describes how SonicMQ can be deployed in very large-scale applications, such as marketplace (portal and trading partners) scenarios, using a multi-node architecture.

This part contains the following chapters:

- Chapter 7, "Dynamic Routing Architecture in a Multi-node Application," describes key elements of the Dynamic Routing Architecture.
- Chapter 8, "Implementing Multi-node Installations," describes the steps you might follow to set up a SonicMQ deployment with portals and trading partners.
- Chapter 9, "Running a Sample Multi-node Application with the Dynamic Routing Architecture," gives step-by-step details on how to set up a demonstration portal and trading partner.

Chapter 7

Dynamic Routing Architecture in a Multi-node Application

There are many applications for dynamic routing, but this guide uses a marketplace as a comprehensive example. This chapter describes key elements of the Dynamic Routing Architecture (DRA) in terms of how they are used to set up a marketplace application. In particular this chapter shows how the elements of the SonicMQ solution relate to such areas as:

- Store & forward queue routing from Trading Partners
- Load-balanced Trading Partner connections
- Load-balancing Portal applications
- Queue routing from Portal to Trading Partners
- Trading Partner configuration
- System management
- Portal management
- Trading Partner management
- Dead message queue
- Trading Partner request/reply example

For the purpose of this example, we use the queue names listed below. These names might differ in your implementation:

- **Portal::appQ** — The name of the queue that is to be handled by a service on the Portal itself. This service can be replicated. Trading Partner applications will write to this queue.
- **TP name::inQ** — The name of the queue that is to receive messages for a particular Trading Partner. Only the Portal itself will be able to connect to this routing node and forward to this queue.
- **TP name::tmpQ** — The name of the queue that is to receive transient reply messages for a particular Trading Partner. The **tmpQ** queue does not have to be a different queue from the **inQ** queue. However, there are reasons to use a separate queue for small, nonpersistent replies to synchronous requests. Having a separate queue facilitates easier maintenance and administration. The **tmpQ** queue can be cleared without losing anything critical, while the **inQ** queue contains messages you cannot afford to lose.

Store & Forward Queue Routing from a Trading Partner

The following components are installed at the Trading Partner:

- A SonicMQ Server set up for Queue Routing
- A SonicMQ administration client (Explorer or Admin)
- One or more Trading Partner Applications

These components can reside on one or more machines and do not have to be running continuously. Only the SonicMQ server is needed to store and forward messages to the Portal.

The Trading Partner would probably have many application users that would be configured in the server database. These users are known only to the Trading Partner server and are not shared with the Portal.

Figure 31 shows how a typical installation at a Trading Partner named Xyz Company communicates with the Portal Application running at the Portal.

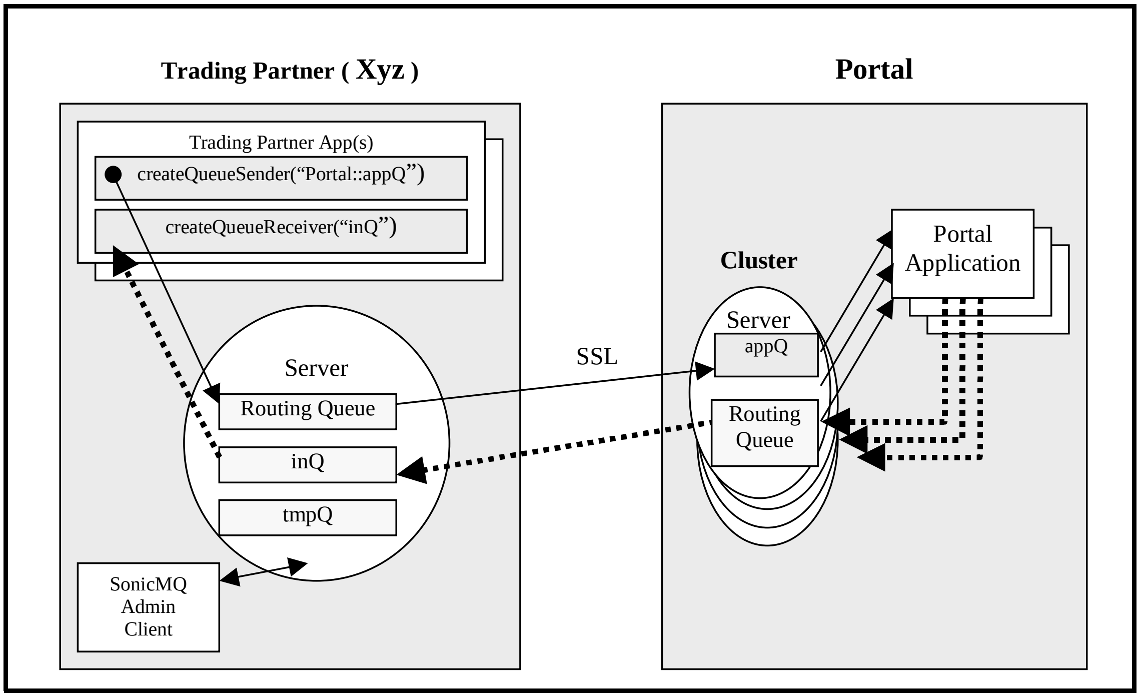


Figure 31. Routing Communication

➤ When a Trading Partner application starts, it performs the following tasks:

- The application connects to the Trading Partner server.
- An application-specific client session is created in the connection.
- An application that wishes to send documents to the Portal must create a QueueSender for the **Portal::appQ** queue.
- To receive documents routed to the Trading Partner, an application must create a QueueReceiver for the global queue, **inQ**. (Applications at the Trading Partner simply connect to the Trading Partner's servers and consume messages using normal calls.)
- A Trading Partner application can be a QueueSender, a QueueReceiver, or both.

Trading Partners cannot communicate directly with one another. All communication is made to the Portal itself. The Portal is the only routing node that can route messages to the Trading Partner.

➤ **When an application at the Trading Partner sends messages to the Portal:**

1. A message is created and populated with application-specific data.
2. A QueueSender is used to send the message to the **Portal::appQ** queue. Both the message and the QueueSender use standard JMS calls to set quality of service and delivery options.
3. The Trading Partner server receives the message in a guaranteed manner subject to the quality of service options used by the sender.
4. Because the **Portal::appQ** is a remote queue name, the message will be placed on the routing queue.
5. The server will process the routing queue by checking the routing node name on each request to see if an active connection exists to this routing node. If not, a new routing connection will be created. (See the section "Load-balanced Trading Partner Connections" on page 113.) This connection will be authenticated using the identity of the Trading Partner installation as a whole, and not using the identity of the original application connection.

   The check described here ensures that a routing node can forward only to adjacent routing nodes. That is, queue routing only applies to a single routing node hop over a routing connection that has been defined as valid by the administrator.
6. One of many similar servers at the Portal will reliably receive the message.

   The Portal now checks if the Trading Partner has permission to send to the **appQ** queue.

   If the message is `PERSISTENT`, it is acknowledged by the Portal.

   The Trading Partner server will confirm the acknowledgement. This acknowledgement confirmation is necessary to guarantee once-and-only-once delivery of `PERSISTENT` messages.

   The message is now removed from the Trading Partner routing queue.

7. The message will be delivered to one of many Portal Applications servicing load-balanced versions of **Portal::appQ**. (See the section "Load-balanced Trading Partner Connections" on page 113).

Load-balanced Trading Partner Connections

There will generally be numerous Trading Partners so that a server cluster will be required at the Portal. The connection to these servers cannot be statically defined. In order to provide scalability, as well as fail-over in the event of system failure, each Trading Partner server can be configured to connect to any one of the Portal servers.

Note The use of both a failover list and load balancing are optional.

➤ **Trading Partner Connections are maintained as follows:**

1. The Portal creates a SonicMQ cluster of servers that can receive connections from Trading Partners.
2. At configuration, each Trading Partner establishes a list of default URLs to connect to the cluster. Typically, this will be a subset of the servers in the cluster and provides a level of initial correction failover. These URLs can be SSL connections (or any other server protocol accepted by SonicMQ). The initial connection to one cluster member is used strictly to return a load-balanced connection to any one of the servers in the cluster.
3. When a message arrives in the Trading Partner's Routing queue, and if there is no routing connection, the Trading Partner server tries to establish a connection to the Portal's cluster.

   SSL is supported for server-to-server communications between routing nodes and can be turned off. Mutual authentication can be specified.
4. An entry in the connection list is read, chosen sequentially or randomly, depending on configuration settings. A connection is made to this server. This connection, however, returns the URL for another server in the Portal's cluster. The choice of URL is based on a round-robin of all the servers currently active in the cluster. As servers are added or removed, the choices for this round-robin automatically change.

5. The Trading Partner connection is made to the redirected server. This new connection is where message traffic actually occurs.
6. The list of initial connect URLs for the routing connection can be updated by the administration tool either locally at the Trading Partner or remotely from the Portal.

The connection from the Trading Partner to the Portal remains active until either:

- The Trading Partner server is shut down
- The Portal server is shut down or the network connection is lost
- The connection times out after an optionally preconfigured idle period

In the event of a failure (if messages remain in the routing queue), the Trading Partner server attempts to reconnect with the connection information it previously used. This "sticky" reconnect is tried for a number of times specified by the `CONNECT_RETRY_COUNT` property. These attempts are spaced `CONNECT_RETRY_INTERVAL` seconds apart. You specify these properties in the `broker.ini` file.

If a connection cannot be re-established after the retry attempts, then after waiting `CONNECT_ATTEMPT_INTERVAL`, updated connection information is retrieved and a connection attempt is made from the configured list of connection URLs. The Trading Partner server continues trying all servers in the list until one of the connections is established.

If all attempts to establish a routing node connection fail, then errors are logged and the process restarts. The new information is retried `CONNECT_RETRY_COUNT` times before the typically longer `CONNECT_ATTEMPT_INTERVAL`. This continues until the specified `ROUTING_TIMEOUT` is reached.

The reconnect logic handles one additional complication. If a server-to-cluster connection fails during the forwarding of some message between a Trading Partner and a Portal, the message might be in an indoubt state. See the "Routing Under Failure Scenarios" section for more information about this situation. From a connection perspective, however, the sending server remembers which messages are indoubt and associates these with a particular connection. Even if a new connection is created between the Trading Partner and the Portal that goes to a different server, attempts are made (subject to a timeout parameter) to re-establish the failed connection and resolve the indoubt state.

That is, indoubt messages will always use the original connection, even if a new connection is used to do routing to a routing node that is a cluster of servers.

For more specifics on connection load-balancing properties, see *Chapter 4, "Failover and Load Balancing."*

Routing Under Failure Scenarios

This section deals with the possibility of failures during the actual process of global queue routing.

There is always a possibility of a network failure or server failure. If this occurs after the sending server has sent a `PERSISTENT` message, but before it has received an acknowledgement, then the message is considered to be in an indoubt state. The message will remain in this state until a connection is re-established between the two servers (or until the `INDOUBT_TIMEOUT` expires).

The sending server will automatically try to re-establish any connections necessary to resolve the state of the indoubt messages. Until this occurs, however, all the indoubt messages will be held where they will not be lost. There is no possibility of message redelivery due to any failure situation, but there is a possibility that the message will take a long time to be delivered.

SonicMQ handles this situation as follows:

- As part of server configuration, a parameter exists that specifies `INDOUBT_TIMEOUT` (in seconds).
- All messages that are held in the indoubt state for a period that exceeds `INDOUBT_TIMEOUT` are automatically expired. You would usually configure all `PERSISTENT` messages to be sent to the **SonicMQ.deadMessage** queue and to raise an administration notification.
- A reason code is associated with messages in **SonicMQ.deadMessage** queue that expire because they are held too long in the indoubt state.
- At no point are these messages lost or inadvertently placed in a state where they can be redelivered.

Messages that are not `PERSISTENT` are not subject to the added overhead of this acknowledgement cycle. This is in keeping with the JMS specification requirement of "at most once" delivery for non-persistent messages.

It is also worth looking at the failure situation from the perspective of the receiving server. As soon as this server receives a message from the sending server, it prepares for guaranteed once-and-only-once delivery, and then sends the acknowledgement back to the originating server. The message will be logged if necessary. A tracking number is created and retained until a confirmation is returned from that server.

If the network or the sending server fails before the receiving server has received confirmation of the acknowledgement, then the receiving server will not discard the acknowledgement. It will retain it until the connection has been re-established. When the two servers reconnect, all unconfirmed acknowledgements can automatically be resent to resolve all inconsistencies.

The important facts about these scenarios are that:

- Messages are never redelivered by SonicMQ queue routing even in the event of network failure.
- Messages can be stored on a sending server in an indoubt state if there is some failure.
- SonicMQ will attempt to re-establish the server-to-server connection to resolve indoubt messages even if another server-to-cluster connection has been created for the destination routing node.
- The destination server will process messages as soon as they have been successfully received and the acknowledgement has been sent.
- The indoubt resolution process uses the SonicMQ journal to retain state. Even if both servers fail in the process at different times, guaranteed exactly once delivery is assured.

A more detailed description of SonicMQ dead message queues can be found in Chapter 3, “Guaranteeing Messages.”

Exchanging Connection Information for Indoubt Resolution

When one server connects to another for global queue routing, the two servers pass information about each other’s routing node between them. Two pieces of this information relate to how the server that received the original connection can reconnect to the originating server.

Note The `DEFAULT_ROUTING_ACCEPTOR` property applies only to indoubt messages. New messages that arrive can trigger a new connection to a different server in the cluster.

Consider the case where a server in a Routing Node (named **Mart**) connects to a portal named **Xchange.** Consider the most complicated case, where both **Mart** and **Xchange** are clusters.

It is clear that **Mart** knows how to connect to **Xchange** (because it made the original connection in the first place). Regardless of whether the initial contact was load-balanced across the **Xchange** cluster, the particular server in **Mart** knows exactly what the ultimate connection properties were that allowed it to connect to a particular server in **Xchange** (server URL, password, username).

If the connection fails, then **Mart** can reuse this information to attempt to reconnect back to the exact same server on **Xchange**. If **Mart** needs to reconnect, it can reuse the old connection information in order to resolve indoubt messages.

However, this does not explain how **Xchange** can reconnect to **Mart** if messages are indoubt on **Xchange.** The reason **Xchange** can reconnect to **Mart** is that **Mart** passed the value of its `DEFAULT_ROUTING_ACCEPTOR` (from its `broker.ini` file) into **Xchange** when the connection was first established.

Advertising Routing Connection Information

The other properties for this connection (username, password, timeout, advertising) are retrieved from the **Xchange** routing connection database. **Xchange** looks up **Mart** and uses only these fields, not the entire connection URL or load-balancing. If there is no entry for **Mart**, then the following defaults are used:

- `username = "AUTHENTICATED"`

  This indicates that the certificate identity will be used in the SSL connection.
- `password = ""`
- `timeout = CONNECT_IDLE_TIMEOUT`
- `advertise = false`

The other piece of connection-related information that passes between **Mart** and **Xchange** when **Mart** connects to **Xchange** is a statically defined "routing incoming-connection." This allows **Mart** to override the outgoing routing connection pre-configured on **Xchange** or create a dynamic value for it if the connection is not present.

Mart creates its routing incoming-connection by entering a routing connection for itself in its own routing connection database. The Administrator on **Mart** can use Admin tool, Explorer, or the Management API to create this entry. For example, the Administrators could issue the following Admin tool commands:

```
[on Mart]:
Admin>set routing Mart "ssl://www.mart.com" user pwd 300 lb
```

This information will only be sent to **Xchange** (but only if **Mart** has used the `advertise` flag on its connection to **Xchange)**. That is, it has

```
[on Mart]:
Admin>set routing Xchange "ssl://xchange.com" user pwd advertise
```

Xchange can choose to ignore this incoming-connection information sent from **Mart** by specifying that it is to use static routings only when going to **Mart**.

```
[on Xchange]:
Admin>set routing Mart "ssl://172.09.3.192" "" "" 4000 nolb static
```

Routing connections are subject to the following conditions:

- `DEFAULT_ROUTING_ACCEPTOR` is always passed. However, advertising the routing incoming-connection is only done if the original routing connection has the `advertise` flag set.
- The incoming-connection information is never used if the routing node has an outgoing connection defined as `static`.
- To enhance security, the routing incoming-connection never passes the username or password across the wire.
- The following parameters of an advertised incoming-connection are never used by the receiving server: username, password, idle timeout, and the advertise flag. These parameters are always retrieved from the outgoing routing connection table.

- If the outgoing routing connection table does not contain an entry for a given routing node, the values for username, password, idle timeout, and the advertise flag are set to:
  - username = `"AUTHENTICATED"`

    This indicates that the certificate identity will be used in the SSL connection.
  - password = `""`
  - timeout = `CONNECT_IDLE_TIMEOUT`
  - advertise = `false`
- If `DEFAULT_ROUTING_ACCEPTOR` is not defined, it will default to a combination of the first (index = 1) acceptor in the `broker.ini` file. That is, it will be:

  `DEFAULT_SOCKET_TYPE``://`*`IP_OR_HOST_1`*`:`*`PORT`*
- The username and password found in the outgoing routing connection table. If these are not found, then `AUTHENTICATED` is used, which implies the use of certificate identity on an SSL connection.

Connection Timeout

The connection from Trading Partner to Portal can be automatically closed by the Trading Partner if the connection has been configured with a timeout (or idle) parameter.

The value of the idle connection timeout in seconds (`CONNECT_IDLE_TIMEOUT`) is specified when a routing for a routing node is defined through the Admin tool, Explorer, or Administration API. The default value of the idle connection timeout is 300 seconds (5 minutes). If the connection idle timeout value is 0, the connection will not timeout.

If a routing connection remains idle for the duration specified by the timeout, it is terminated. A message describing this event is displayed on the servers at both ends of the connection. When the next message arrives in the routing queue for this server new load-balanced connection will be made (using the connection logic described above).

Each side of a connection between routing nodes independently monitors the idle time, and either one can terminate the connection. Each uses its own timeout, as specified in the routing connections database. If there is no entry for the remote routing node, the `CONNECT_IDLE_TIMEOUT` in the `broker.ini` file will be used. If this value is also not set, it defaults to 300 seconds.

If the idle timeout value is changed for a remote routing node and there are active connections between the local routing node and the remote routing node, the new value takes effect immediately (on the local side of the connection). However, if you change an idle timeout value dynamically, a new timeout period is started.

Portal-initiated Connections

Often the Portal will be required to send messages to the Trading Partners. In the event that the Trading Partner has timed-out its connection, however, the Portal will be required to re-establish the connection.

The Portal (or any server in the cluster) shares routing information by queue name. Each Trading Partner's global queue, such as **Xyz::inQ**, is associated with the connection that the Trading Partner had previously established.

In the event that no previous connection has been created, then the Portal administrator must have created a list of routing connections that associate the individual routing node names to particular server URLs (or lists of URLs).

If a message needs to be sent to the Trading Partner, the route table is used to see how to re-establish the connection, if necessary. If the connection currently exists, the message will be routed to the server in the cluster where the connection is active. If the connection does not exist, the routing connection table is queried for its current value of the connection URLs and other connection properties. This is preconfigured administratively using the Admin tool, Explorer, or the Management API.

If no connection URL has been preconfigured and no incoming-connection has been advertised, then the message will be flagged as undeliverable and checked to see if it will be sent to the `SonicMQ.deadMessage` system queue. Administrative notifications might also be sent.

A Trading Partner can also specify a **Connect URL** to use for incoming routing connections. This is specified for the server and will be advertised to the Portal when the Trading Partner first connects. If this advertised connect URL is specified, then it will be used before the connection URLs statically configured on the Portal, unless the Portal overrides this setting by using the **static** flag on its preconfigured routing connection.

There might be some cases where a Trading Partner is unwilling to expose its routing node to external connections. In this case, the Trading Partner must maintain the connection to the Portal without a timeout. Otherwise, if the connection times out, messages destined for the Trading Partner will be lost.

Connection Security

The connection between two servers will be mutually authenticated. Server-to-server connections can occur when two routing nodes connect or when servers mutually connect within a cluster. In both cases, SSL is optional.

Note: SSL support is only available with SonicMQ Professional Developer Edition and E-Business Edition.

Table 8 indicates what authentication is done on a connection by a SonicMQ server when a client or another server attempts to get a connection.

Table 8. Connection Security Checking

Password-based Authentication	***SSL Client Authentication***	***Behavior***
disabled	disabled	The client connection is always accepted.
disabled	enabled	Only valid chains that contain a trusted certificate are accepted. Valid means that all signatures verify and no certificates are expired.
enabled	disabled	The client connection is accepted only if the client is successfully authenticated by username/password. This authentication mode is typically used in multi-node applications, between individual clients and their local servers.

Table 8. Connection Security Checking (*continued*)

Password-based Authentication	*SSL Client Authentication*	*Behavior*
enabled	enabled	Only valid chains that contain a trusted certificate are accepted. Valid means all signatures verify and no certificates are expired. Once a client certificate chain is accepted, the client connection is accepted only if the client is recognized by the server. This is done by verifying that the username for the connection matches the principal name in the security database. If the username is the string **`AUTHENTICATED`** ,the effective username will be retrieved from the identity embedded in the client certificate (the Subject Common Name). When using this certificate identity, the user is automatically authenticated without checking passwords. This authentication mode is typically used in connections between servers in different routing nodes.

Load-balancing Across Portal Applications

Your Portal will probably contain a collection of replicated services. Chief among these could be the main Portal Application whose task is to examine incoming messages from a Trading Partner and route these to other Trading Partners.

The tasks of the Portal Application are to:

- Receive a single request to the **Portal::appQ** queue.
- Look at the manifest.
- Decide where the message needs to be routed.
- Update the manifest if necessary and send the message to a Queue associated with some other Trading Partner.

To accomplish this processing, you might use one or more of the following features:

- You can make both the receive and the send part of a single local client transaction. That is, a set of messages might need to be committed as a single logical unit of work.
- The processing of routing requests might need to be totally stateless. In this case, each active application should **not** be dedicated to a particular Trading Partner.
- Portal Applications can be located on remote computers. Each application can connect to many or all servers in the cluster.

The ability of the Portal to load-balance dynamically between the Portal Applications happens automatically based on standard JMS QueueReceiver behavior:

- Each Portal Application creates connections to one or more servers in the Portal cluster.
- Each of these connections is used to create a transacted JMS QueueSession.
- The QueueSession is used to create a QueueReceiver on the queue **appQ,** which exists on each server in the cluster.
- A QueueSender is also created on each of these QueueSessions.
- This load-balancing across QueueReceivers on the same queue occurs automatically in SonicMQ. Load-balancing of Portal services is not configured at the server/cluster level.

Figure 32 illustrates the concept of load balancing.

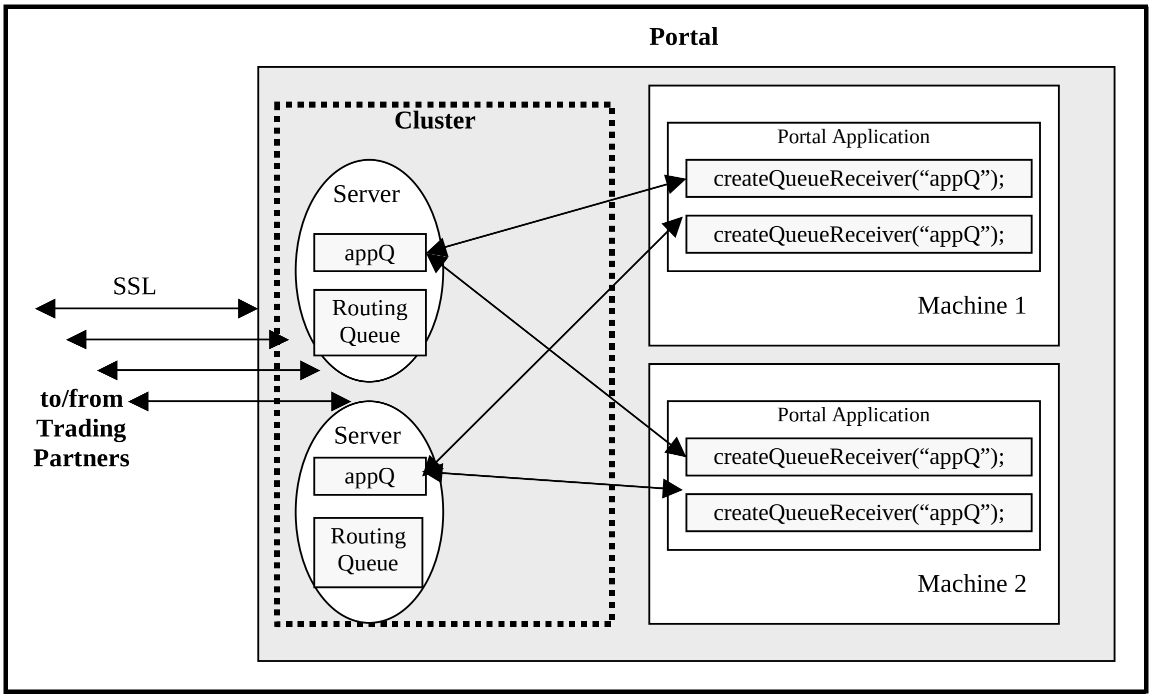


Figure 32. Routing: Load Balancing

When messages arrive at a cluster destined for a particular queue, they are automatically routed to the nearest server that supports this queue. This means that if the actual server where the connection is made supports this global queue, the message will be delivered there. Only if the receiving server does not support the queue directly will it be routed to another server in the routing node where this global queue is defined.

Queue Routing from Portal to Trading Partners

In routing from the Portal Application to a Trading Partner, the Portal Application simply needs to identify the appropriately named queue for that partner. Figure 33 illustrates routing from a Portal Application to a Trading Partner.

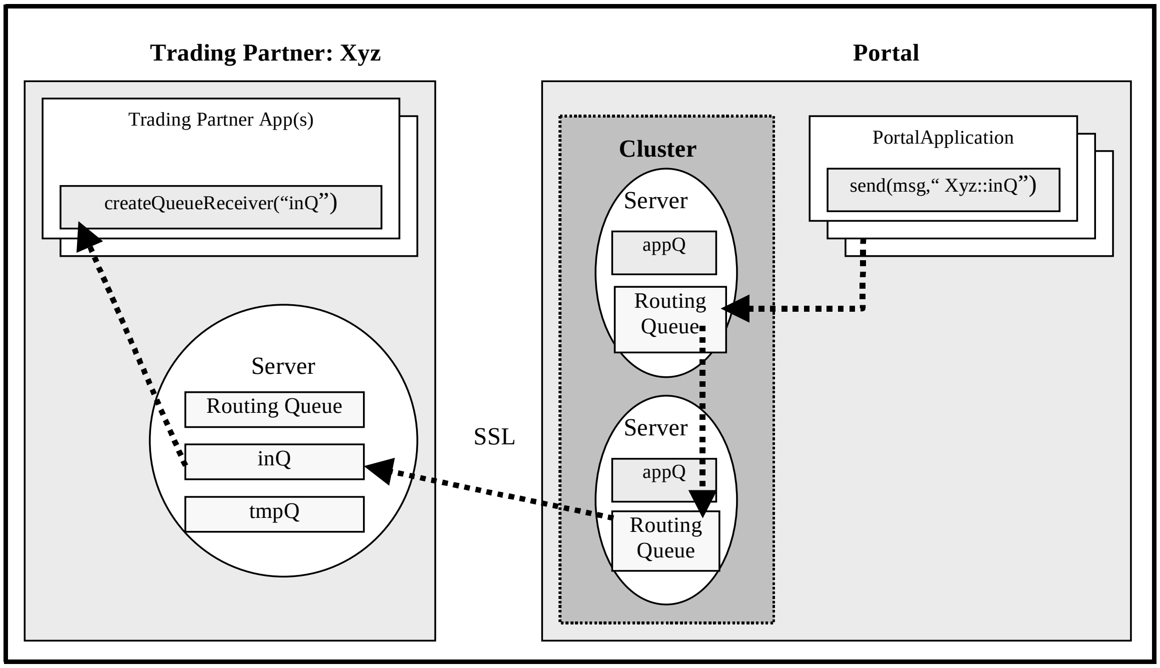


Figure 33. Routing: Portal to Partner

➤ To send a message from a Portal Application to a queue on a Trading Partner:

1. The Portal Application needs to send to a particular Trading Partner so it finds the associated routing node name for that Trading Partner. In the example, the name is **Xyz**.
2. The routing node name is combined with the name of the queue to create the remote queue name (**Xyz::inQ**). The Portal Application creates a JMS queue destination for this name and sends it to a server in the cluster.

3. The clustered server looks up this destination in its route table.

   The connection for the **Xyz** routing node exists in the route table because the last time that Trading Partner XYZ connected, it advertised this queue to the cluster as a global queue that it supports. The route table persistently remembers this connection.

4. When the router needs to forward to **Xyz::inQ** it looks up the connection in the route table:

   - If the connection is still active, it is used. This step might involve a hop where one server in the cluster routes the message to the server that is connected to **Xyz**.
   - If the connection has timed-out, become idle, or just closed, the routing server will attempt to re-establish the connection with the Trading Partner server.

5. When the message does arrive at the Trading Partner server, it is immediately placed on the **inQ** queue.

6. An existing application that has created a QueueReceiver on the **inQ** queue will receive the message (either synchronously or asynchronously) as programmed using normal design patterns.

Access control is maintained in the Portal configuration where the username of a Trading Partner is uniquely mapped to the name of its routing node. No other Trading Partner can receive messages for the routing node associated with a given Trading Partner. Because the username is determined from the certificate used to create the SSL connection, this level of security ensures that no Trading Partner can accidentally or intentionally intercept messages intended for another Trading Partner.

System Management

The SonicMQ multi-node solution supports centralized management of the Portal and the Trading Partners. It also lets individual Trading Partners manage their resources locally.

You can perform all management tasks one of three ways:

- Admin tool, a command-line tool that supports the use of scripts. See the *SonicMQ Installation and Administration Guide* for a description.
- SonicMQ Explorer, a graphical interface tool. See the *SonicMQ Installation and Administration Guide* for a description.
- The Management API. See `progress.message.tools.BrokerManager` in the javadoc.

All three options require you to connect to the server with an administration name and password. The connection is made through JMS, can be done remotely, and can be secured using SSL.

Portal Management

The Portal is usually a cluster of servers, in which case you should manage all servers using a configuration server. By using a configuration server you can update information once for the entire Portal.

Trading Partner Management

You can manage Trading Partner servers in three ways:

- Use Admin or Explorer tools from the Trading Partner itself.
- Use Admin or Explorer tools from the Portal to allow for remote configuration of the Trading Partner by the Portal operators.

Note This is optional. Trading Partners may not allow remote administration of their server from the Portal.

- Have Trading Partner applications or applications running at the Portal site manage the servers using the Management API.

For example, as Portal configuration changes or Trading Partner information is updated, these applications can automatically reconfigure the server information at the Trading Partner. This could only be done with permissions granted by the Trading Partner.

Dead Message Queue

Because of hardware or network failures, or for other reasons, it is always possible that a message will fail to be delivered. When this happens, an administrative event is generated and (depending on a setting in the message header) the message might be sent to a special system queue called the Dead Message Queue. See Chapter 3, “Guaranteeing Messages” for a thorough description of this subject.

Trading Partner Request/Reply Example

This section presents an example showing how applications at the Trading Partner can implement a synchronous request/reply layer on top of global queue routing.

Normal support for synchronous request/reply design patterns is complicated in the case of global queue routing due to the following issues:

- Creating unique temporary queues
- Accessing temporary queues across Trading Partner security domains

Typically, the design pattern for request/reply is to:

- Make a temporary queue
- Set the JMSReplyTo header to this destination
- Do a synchronous `QueueSender.receive()` on the message (with an optional timeout)

The JMSReplyTo header is likely to be used when a Trading Partner application needs some low-latency synchronous interaction with other partners. This might be a quick price check, inventory status, or similar type of information. For this type of request, the application is expected to be blocking for less than

10 seconds or so. The developer should use messaging with the following settings for synchronous requests:

- Low Quality of Service (`NON_PERSISTENT`, unencrypted, small messages).
- Explicit, and short, Time To Live.
- Message expiration might raise a notification, but the message will not be saved in the Dead Message Queue.
- High priority (to expedite delivery).

If the request is lost, the application is expected to simply retry the request.

The issue with the normal design pattern of using a temporary queue for this interaction is the need to prevent a Trading Partner from knowing about another's requests. Instead of a temporary queue, it is more convenient to use a second global queue at the Trading Partner's site. This second queue can be configured at the Portal to ensure security.

In our examples, this queue is named **tmpQ.** The reason this queue is not the same as the normal **inQ** queue is to allow for easier administration. Messages on **tmpQ** can be assumed to be transient, and the queue can be cleaned up without worrying about losing important business documents.

Many applications at the Trading Partner might simultaneously request information. Because of security concerns, it is easier for them to share a single queue. Each application can use selectors on its QueueReceiver to request its reply.

Here is sample pseudo-code that illustates this use of selectors:

```
// Create a request
TextMessage m = session.createTextMessage();
m.setJMSReplyTo(“acme::tmpQ”); // psuedo-code
m.setText(“Inventory Check: #1234”);
// Create a unique queue receiver for the reply
// NOTE use of selector
String uniqueID = createUniqueId();
m.setProperty(“AppUniqueId”, uniqueID);
QueueReceiver qr = session.createQueueReceiver
        (“tmpQ”, “AppUniqueID = ‘” + uniqueID + “’”);
// Wait 7 seconds for a reply.
TextMessage rep = qr.receive(m, 7000);
```

Chapter 8

Implementing Multi-node Installations

Introduction

This chapter describes the SonicMQ setup needed to implement the Trading Partner and Portal installations described earlier. Figure 34 illustrates the relationship between the Trading Partner and Portal.

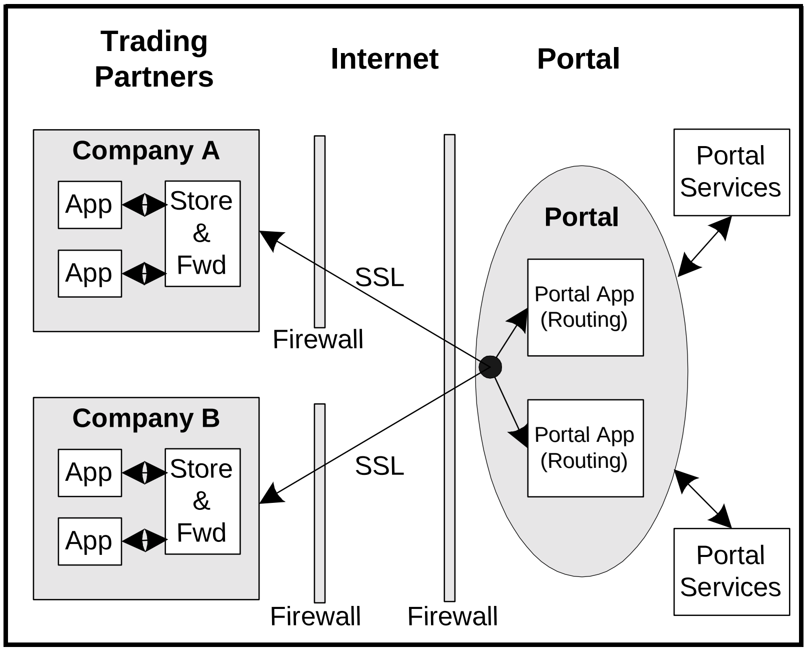


Figure 34. High-level View of Trading Partner-Portal Configuration

In particular, this chapter addresses:

- Trading Partner Configuration:
  - Firewall setup
  - SonicMQ server installation and setup
  - SonicMQ routing node setup
  - SonicMQ security/routing configuration
- Portal Configuration:
  - Firewall setup
  - SonicMQ server installation and setup
  - SonicMQ configuration server installation and setup
  - SonicMQ routing node setup
  - SonicMQ security/routing configuration

Important See the *SonicMQ Installation and Administration Guide* for more information, especially in the areas of security and server cluster configuration.

Definition of Terms

Table 9 provides definitions for the names used in the sample configuration files and Admin Shell scripts that are presented later in the chapter to implement the configurations. You would change most of these names when deploying your own SonicMQ multi-node configuration.

Table 9. Names Used in Sample Admin Shell Scripts

Name	*Definition*	*Location*
Acme	The routing node name for the sample Trading Partner	Acme's `broker.ini`
AcmeCo	The routing username Acme uses to connect to Xchange	Security database at portal (ntconfig)
Applications	The group name for administering security for all application users as a group (user1, user2, and user3 are members of this group)	Security database at Acme
appQ	The global queue that is to be handled by the portal routing application	Every portal server
ConfigServer	The `BROKER_NAME` for the configuration server at the portal	`broker.ini` on ntconfig
direct.A	The global queue on portala.xchange.com that handles synchronous requests	On server PortalA (at portal)
direct.B	The global queue on portalb.xchange.com that handles synchronous requests	On server PortalB (at portal)
direct.C	The global queue on portalc.xchange.com that handles synchronous requests	On server PortalC (at portal)
inQ	The global queue that is to receive messages for a particular Trading Partner	Acme's server
ntconfig	The machine name for the configuration server (not accessible from outside the portal by a DNS lookup)	“Hosts” file available to all portal machines
PortalA	The `BROKER_NAME` for one of the clustered servers at the portal	`broker.ini` on one portal machine
portala.xchange.com	The name of the machine hosting one of the portal’s publicly accessible servers	DNS lookup server on the Internet

Table 9. Names Used in Sample Admin Shell Scripts (*continued*)

Name	***Definition***	***Location***
PortalB	The `BROKER_NAME` for one of the clustered servers at the portal	`broker.ini` on one portal machine
portalb.xchange.com	The name of the machine hosting one of the portal’s publicly accessible servers	DNS lookup server on the Internet
PortalC	The `BROKER_NAME` for one of the clustered servers at the portal	`broker.ini` on one portal machine
portalc.xchange.com	The name of the machine hosting one of the portal’s publicly accessible servers	DNS lookup server on the Internet
PortalCluster	The name of the cluster at the portal	Cluster database at portal (ntconfig)
pUser	The user name used by portal routing applications	Security database at portal (ntconfig)
Routing Queue	The internal “routing queue” automatically created on all servers	All SonicMQ servers
RoutingUsers	The group name for administering security for all portal routing applications as a group (pUser is a member)	Security database at portal (ntconfig)
thePortal	Routing user name for the Xchange routing node when it connects to Acme	Security database at Acme
TradingPartners	The group name for administering security for all trading partners as a group (AcmeCo is a member)	Security database at portal (ntconfig)
user1	Sample user name for application users at Acme	Security database at Acme
user2	Sample user name for application users at Acme	Security database at Acme
user3	Sample user name for application users at Acme	Security database at Acme

Table 9. Names Used in Sample Admin Shell Scripts (*continued*)

Name	*Definition*	*Location*
www.acme.com	The name of the machine hosting Acme's SonicMQ server (accessible by external DNS lookup)	DNS lookup server on the Internet
Xchange	The routing node name for the portal	Each `broker.ini` for all clustered portal servers

High-level Architecture

Figure 35 illustrates how a typical installation at a Trading Partner (named Acme) communicates with the Portal Application running at the portal (named Xchange).

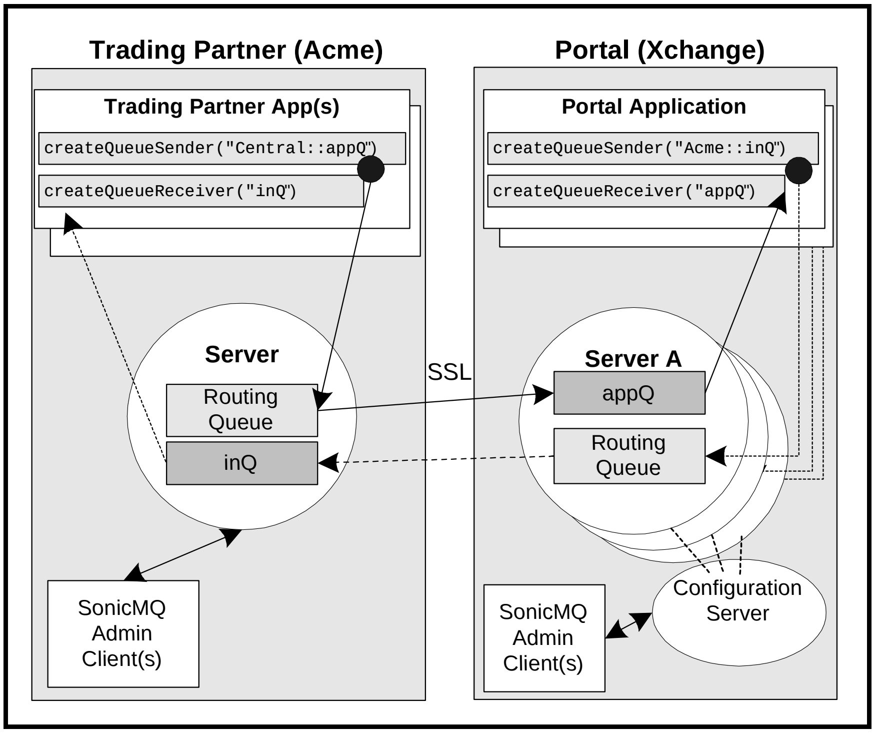


Figure 35. Trading Partner-Portal Configuration

The following sections discuss this setup in greater detail.

Trading Partner Configuration

This section describes the configuration of a Trading Partner named Acme Company. This company has installed their SonicMQ server in the DMZ on a machine accessible as www.acme.com. As part of their agreement with the Xchange Portal, Acme Company has committed to:

- Install the SonicMQ server on www.acme.com.
- Punch a hole in the outside firewall to allow SSL connections on port 2507.
- Allow incoming connections from Xchange on port 2507. The Xchange user must identify itself as **thePortal** over this connection.
- Use the routing node name **Acme** in dealings with Xchange.

When connecting to the Portal, Acme Company has agreed to:

- Connect using the Portal contact points: `ssl://portala.xchange.com:2507` and `ssl://portalb.xchange.com:2507`.
- Identify itself using a certificate supplied by the Portal. This certificate contains the username **AcmeCo**.

Acme Company also plans to use Trading Partner applications that will access the local server on www.acme.com:

- AcmeCo will create a group of users called **Applications**.
- The users in this group, **user1**, **user2**, and **user3**, will be managed by Acme.
- No users except **user1**, **user2**, and **user3** can send to the **appQ** at **Xchange** or can read from the **inQ**.
- These users must be able to access the Acme Company server. Acme Company has decided to use normal TCP connections to the server. Their internal firewall will be configured to allow for connections to `tcp://www.acme.com:2506`.

Acme Company has refused to allow the Xchange Portal to remotely administer their installation:

- Acme Company will not configure a **RemoteXchangeAdmin** user (in the Administrators group).
- The Administrators group will have a single user, **Administrator**, set up locally.

Figure 36 shows how the Acme installation will look.

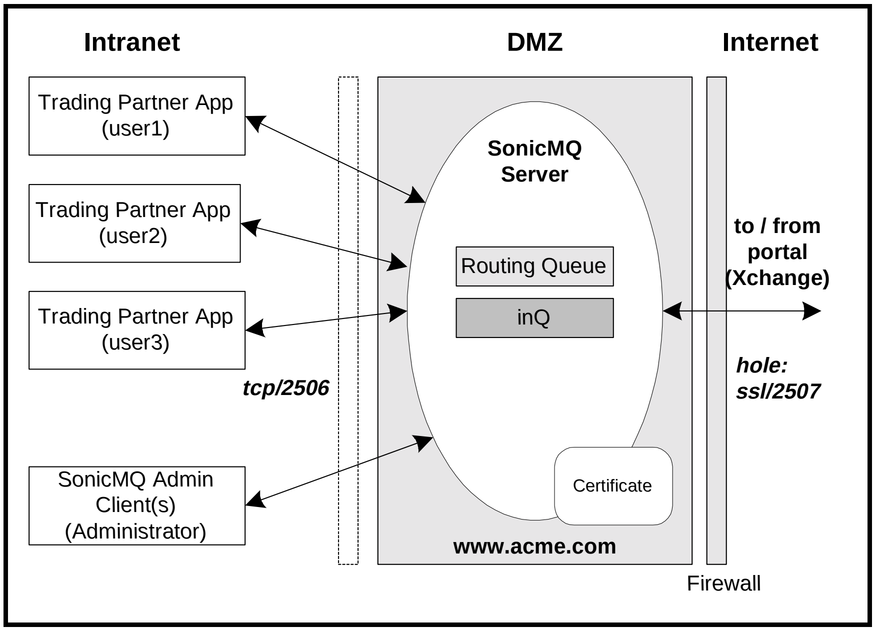

Intranet
DMZ
Internet
Trading Partner App (user1)
Trading Partner App (user2)
Trading Partner App (user3)
SonicMQ Admin Client(s) (Administrator)
tcp/2506
SonicMQ Server
Routing Queue
inQ
Certificate
www.acme.com
to / from portal (Xchange)
hole: ssl/2507
Firewall

Figure 36. Trading Partner Configuration for Acme Installation

Firewall Setup

The firewall setup requires the following steps:

- Configure the DMZ to allow an SSL connection over port 2507.
- Configure internal applications to allow TCP connection over port 2506.

SonicMQ Trading Partner Configuration

The following steps tell you how to configure a trading partner.

➤ Standard SonicMQ configuration for all Trading Partners:

1. Install SonicMQ.
2. Re-create the database using the SonicMQ dbtool to reconfigure the basic tables and create the security database:

   ```
   c:\sonic\bin> dbtool /r basic
   c:\sonic\bin> dbtool /c security
   ```

3. Change the server log filename (recommended):

   ```
   BROKER_LOG=server.log
   ```

4. Make any other changes to the `broker.ini` file that you wish. For example, you might wish to make the server part of a cluster.

   More information on standard `broker.ini` settings is available in the *SonicMQ Installation and Administration Guide*.

SonicMQ Static Configuration

After you have completed the installation at a Trading Partner, you must set parameters to define the specifics of this installation. Set the parameters in `broker.ini` as follows:

- Global Routing Parameters:
  - Set `ROUTING_NODE_NAME`, the unique, maximum 256-character, routing node name for this portal.

    In this example, set `ROUTING_NODE_NAME=Acme`.
  - Set `DEFAULT_ROUTING_ACCEPTOR`, the preferred URL for indoubt connections. This parameter is optional, but useful if there are any issues with how the Routing Node is to be exposed to the Internet. SonicMQ attempts to create this by default based on the IP address of the node and its default socket/port, that is, `ssl://www.acme.com:2507`.

- SonicMQ Acceptors:
  - Set the `NUM_ACCEPTORS to` specify how many access modes there will be. For each mode you must also specify the IP address or name, socket type (SSL or TCP), and port.
  - For each acceptor, specify the socket type and port.
  - For each SSL acceptor, specify the SSL certificate information (see below).

    In this example these settings are:

    ```
    NUM_ACCEPTORS=2

    DEFAULT_SOCKET_TYPE=ssl
    IP_OR_HOST_1=www.acme.com
    PORT=2507

    SOCKET_TYPE_2=tcp
    IP_OR_HOST_2=www.acme.com
    PORT_NUMBER_2=2506
    ```
- SSL must be configured to support certificates and encryption:

  ```
  SSL_CLIENT_AUTHENTICATION=TRUE
  SSL_CIPHER_SUITES=SSL_RSA_WITH_3DES_EDE_CBC_SHA
  SSL_PRIVATE_KEY=certs/serverkey.der
  SSL_PRIVATE_KEY_PASSWORD=your_password
  ```

  This configuration assumes the default values of the following SSL settings:

  ```
  SSL_CA_CERTIFICATES_DIR=certs/ca
  SSL_CERTIFICATE_CHAIN=certs/serverCertChain.chain
  ```

SonicMQ Admin Configuration

After the preceding configurations are complete, you must establish queues, users, and groups. You can complete the following tasks with the Admin shell script provided in this section.

You can also use Explorer or the Administration API:

1. Initialize the server configuration for those items needed to support the Trading Partner application:
   - Application queues/topics
   - Application-related users/admin

2. Put the users in an **Applications** group and give them the following access rights:
   - SEND to appQ
   - RECEIVE to inQ
3. Give the Portal routing user access rights to:
   - SEND to inQ (and any other application queues)

The following Admin shell script shows the commands necessary for setting up the Portal:

```
//
// Setup file for Trading Partner
//

connect broker localhost Administrator Administrator

// Create local application users (in an "Applications" group)
// Add sample users to the group

add group Applications
add user user1 pwd1
add user user2 pwd2
add user user3 pwd3

add groupuser Applications user1
add groupuser Applications user2
add groupuser Applications user3

// Portal is at "ssl://portala.xchange.com:2507". Use a load
// balanced connection with a 10 minute timeout.


set routing Xchange "ssl://portala.xchange.com:2507" /
AcmeCo pwd lb 600 advertise

// Configure the Portal user (to allow "Xchange" to
// call back into this routing node)

add routing user thePortal "Xchange" pwd

// Create incoming global queues.

set queue inQ global

// Override properties to the system queues.

set queue SonicMQ.deadMessage local 1200, 1400, 15000
set queue SonicMQ.routingQueue local 1200, 1400, 2000

// ACLs -- Prevent everyone from accessing all Queues
// unless explicitly granted below.

set queue acl # PUBLIC -snd -rcv
```

```
// ACLs -- Applications can send to appQ and direct queues
// (on Xchange), and receive from others queue

set queue acl appQ   Applications +snd -rcv
set queue acl direct.* Applications +snd -rcv
set queue acl inQ    Applications -snd +rcv

// ACLs -- the portal user can only send to inQ/tmpQ

set queue acl inQ    thePortal    +snd -rcv

// logout

bye
```

4. Change the Administration password for this installation.

Portal Configuration

The setup of the Portal is more complex than that of the Trading Partner because there is a cluster of servers, and all the possible trading partners must be configured in the security and routing database.

Figure 37 shows a typical Portal configuration.

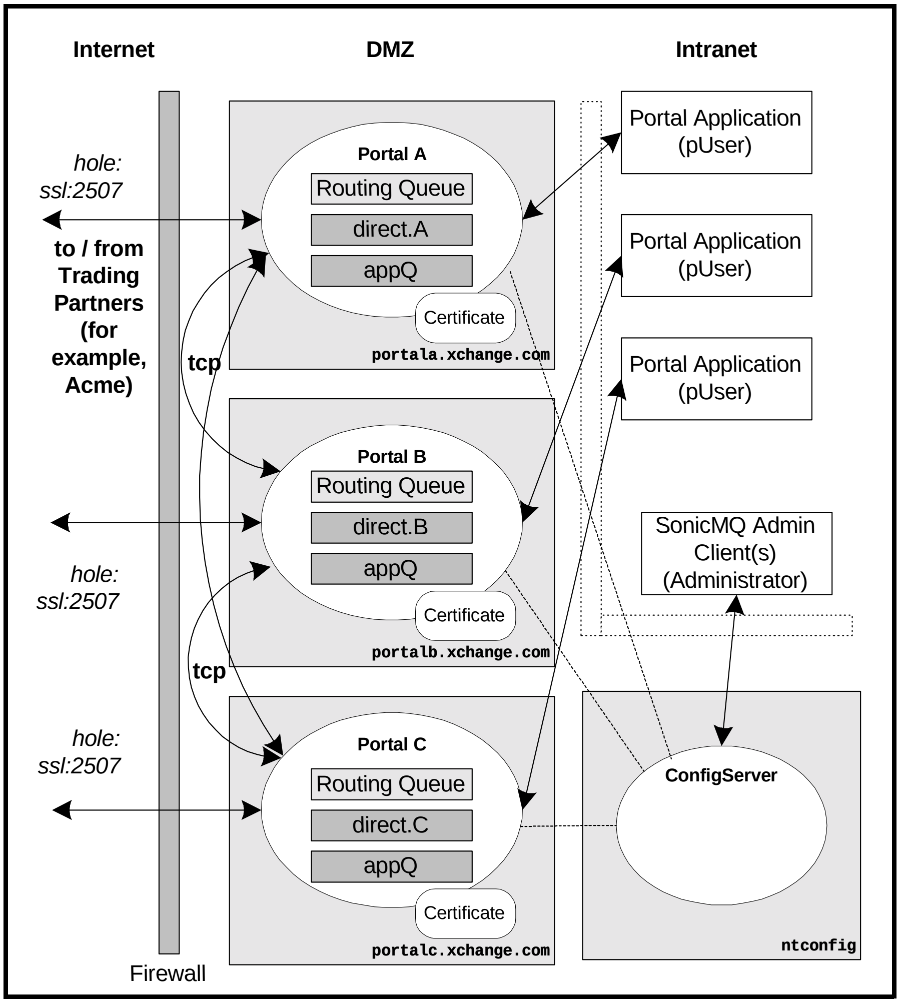


Figure 37. Typical Portal Configuration

In Figure 37, note that the configuration server (ConfigServer) is on a separate machine from all the clustered servers in the DMZ. The machine does not have to be accessible to the Internet so it is named locally as **ntconfig**. This server is set up with `BROKER_NAME=ConfigServer`. The server manages the cluster **PortalCluster**.

Administration is done only on the configuration server.

All Portal Applications use the same username, **pUser**. The Portal Applications connect with unique ConnectIDs to avoid conflict.

There are three machines shown as part of the cluster. Their SonicMQ server names are **PortalA**, **PortalB**, and **PortalC**. These servers are exposed to the Internet using the host names: `portala.xchange.com`, `portalb.xchange.com`, and `portalc.xchange.com`.

Each of these servers supports the **appQ** as a global queue, but they also support a uniquely named **direct access queue** that enables applications to address messages back to this server. (For example, **PortalB** supports **direct.B** as its global queue. Any application connected anywhere to the portal can address messages to **Xchange::direct.B** and the message will be routed to the correct server in the cluster.)

The names of the synch queues, **direct.A**, **direct.B**, and **direct.C**, include periods to allow for wild card ACLs, as shown in the following example:

```
set queue acl direct.* pUser +snd
```

Firewall Setup

The following steps tell you how to configure the DMZ and internal applications to set up a firewall for your portal.

➤ **To set up the firewall for the Portal:**

1. Configure the DMZ to allow an SSL connection over port 2507 for each computer that is accessible from Trading Partners.
2. Configure internal applications to allow SSL connection over port 2508.
3. Server cluster communication (between servers in the cluster) is assumed to be over TCP (port 2506).

Configuration Server Setup

You must set up the configuration server with the `broker.ini` settings shown in the following procedure.

➤ To set up the Configuration Server:

1. Install SonicMQ.

   Make sure the server name is `ConfigServer`. This will set up the correct database tables, as well as set the value for `BROKER_NAME=ConfigServer.`

2. Recreate the database using the SonicMQ dbtool to reconfigure the basic tables, create the security database, and include the server cluster configurations:

   ```
   c:\sonic\bin> dbtool /r all
   ```

3. Change the server log file name (recommended):

   ```
   BROKER_LOG=config.log
   ```

More information on setting up configuration servers is available as part of the *SonicMQ Installation and Administration Guide*.

Clustered Server Setup

You must set up each server to be clustered with a unique name.

Note You must enter the names of clustered servers as user names in the configuration server's security database, and you must add those user names to the Administrators group.

➤ To set up PortalA:

1. Install SonicMQ.

   1.1 Make sure the server name is `PortalA`. This will set up the correct database tables and set the value for

   ```
   BROKER_NAME=PortalA
   ```

2. Recreate the database using the SonicMQ dbtool to reconfigure the basic database tables:

```
c:\sonic\bin> dbtool /r basic
```

3. Set up each server with the same routing node name, but with the default routing acceptor specific to its host:

```
ROUTING_NODE_NAME=Xchange
DEFAULT_ROUTING_ACCEPTOR=ssl://portala.xchange.com:2507
```

4. Set up acceptors for PortalA. Typically, you would set up each server in the cluster with three acceptors: one for connections from the external trading partners (using SSL and port 2507), one for server cluster communications (using TCP and port 2506), and one to be used by Portal Applications (using SSL and port 2508). Use the following property settings:

```
NUM_ACCEPTORS=3

DEFAULT_SOCKET_TYPE=ssl
IP_OR_HOST_1=portala.xchange.com
PORT=2507

SOCKET_TYPE_2=tcp
IP_OR_HOST_2=portala.xchange.com
PORT_NUMBER_2=2506

SOCKET_TYPE_3=ssl
PORT_NUMBER_3=2508
IP_OR_HOST_3=portala.xchange.com
```

5. Set up the second acceptor for server cluster communications with the following server cluster settings:

```
ENABLE_INTERBROKER=TRUE
IB_CONFIG_SERVER=tcp://ntconfig:2506
INTERBROKER_ACCEPTOR=2
```

More information on setting up clustered servers is available in the *SonicMQ Installation and Administration Guide*.

Setting Up Global Queues in a Cluster

You must configure each of the servers in the cluster to support the appropriate global queues. (The access control to these queues is set in the security database maintained by the configuration server.)

For example, you can use the following script to create the queues for the server PortalB:

```
//
// Setup file for each server in the portal cluster.
//

connect broker portalb.xchange.com Administrator Administrator

// Create incoming global queues.
set queue appQ global
set queue direct.B global

// Override properties of system queues.
set queue SonicMQ.deadMessage local 1200, 1400, 20000
set queue SonicMQ.routingQueue local 1200, 1400, 4000

// logout

bye
```

Note You must specify the appropriate incoming global queue for each server in the cluster when creating incoming global queues. For the server PortalB in the preceding case, the command is: `set queue direct.B global`.

Configuration Server Security Configuration

After the preceding configurations are complete, you must establish groups, users, and access control permissions in the configuration server's security database. You can use the Admin tool, Explorer, or the Management API. You can follow these rules to simplify the process:

- All trading partners are part of a larger **TradingPartner** group.
- This group can send to the global queues (**appQ**, **direct.A**, etc.) on the Portal.
- The Portal Applications will all log in as a common user (**pUser**). They might use separate user names, but for this example add them into a single group: **RoutingUsers**.
- Portal members of **RoutingUsers** have access rights to:
  - RECEIVE from **appQ** queue
  - RECEIVE from **direct.*** queues
  - SEND to **inQ**

The following Admin shell script shows the commands necessary for setting up the Portal:

```
//
// Setup file for Portal security database
//

connect broker ntconfig Administrator Administrator

// Create a group for all the internal portal applications.
// Add the necessary local users to this group

add group RoutingUsers
add user pUser pwd
add groupuser RoutingUsers pUser

// Create a group for all the trading partners
// (routing users will be added to this group, later).

add group TradingPartners

// ACLs -- Prevent everyone from accessing all Queues
// unless explicitly granted below.

set queue acl # PUBLIC -snd -rcv

// ACLs -- Portal Applications can receive from appQ/direct queues
// and send to the trading partners inQ/tmpQ

set queue acl appQ   RoutingUsers -snd +rcv
set queue acl direct.* RoutingUsers -snd +rcv
set queue acl inQ    RoutingUsers +snd -rcv

// ACLs -- the TradingPartners can only send to appQ/direct queues
set queue acl appQ   TradingPartners +snd -rcv
set queue acl direct.* TradingPartners +snd -rcv

// logout

bye
```

Portal Configuration for Adding a New Trading Partner

You are now ready to create a new Trading Partner and configure it in the security and routing databases for the Portal cluster. This sample creates the information for **Acme** at `ssl://www.acme.com:2507`.

➤ **To add a new Trading Partner (Acme) at the Portal:**

1. Add the routing user **Acme**. The user name is **AcmeCo**. By adding **AcmeCo** to the **TradingPartners** group, you automatically give it the correct access permissions.

2. Add the routing connection for the Acme site. The routing node name is **Acme**.

   This configuration can be done with the SonicMQ Explorer, Admin tool, or Management API. The following Admin shell script shows the commands necessary for setting up the Portal:

   ```
   //
   // Setup file for adding user "Acme" to the portal.
   //

   connect broker ntconfig Administrator Administrator

   // Create the new user in the TradingPartners group.
   add routing user AcmeCo Acme pwd
   add groupuser TradingPartners AcmeCo

   // Set up the routing to Acme using the username/password
   // that is expected by Acme's security database.
   // Use a 5 minute timeout and a direct (not load-balanced)
   // connection.

   set routing Acme "ssl://www.acme.com:2507" thePortal pwd 300 /
   advertise static

   // logout

   bye
   ```

You have now completed the configuration of your Trading Partner and Portal installations.

Chapter 9

Running a Sample Multi-node Application with the Dynamic Routing Architecture

Introduction

This chapter walks you through a sample setup procedure for a possible scenario using SonicMQ's Dynamic Routing Architecture. This chapter does not cover all the features available in SonicMQ, but rather demonstrates a particular application of the Dynamic Routing Architecture with SonicMQ. You will use scripts and sample programs to set up global routing between a Trading Partner and a Portal. This example is only one of a variety of applications of global routing intended to provide a specific example of the implementations discussed in more detail in Chapter 8, "Implementing Multi-node Installations."

Note The example in this chapter uses security features and global routing concepts. If you are getting started for the first time with SonicMQ, see the samples provided in *Getting Started with SonicMQ*, which do not enable security features or apply global routing concepts.

This chapter shows you how to do the following:

- Create a Portal
- Create a Trading Partner
- Define default routing between these routing nodes
- Use the SonicMQ GlobalTalk sample application across these nodes

Scripts are provided to help you set up your Portal and Trading Partner. These scripts are printed at the end of this chapter.

Assumptions

In this example you will install SonicMQ on two machines. For simplicity, this example specifies the install directory `c:\sonic` on both machines. This example provides the commands for Windows operating systems. You can perform this example on Unix operating systems by following the same steps but using the equivalent Unix commands.

Note If you want to change the install directory for your installation, you must make the appropriate changes in the commands that follow.

This example includes a Portal named **Xchange** and a Trading Partner named **Acme**. The steps show you how to install the Portal Xchange on a machine named **ntportal** and the Trading Partner Acme on a second machine named **ntacme**.

Figure 38 shows the completed installation configuration:

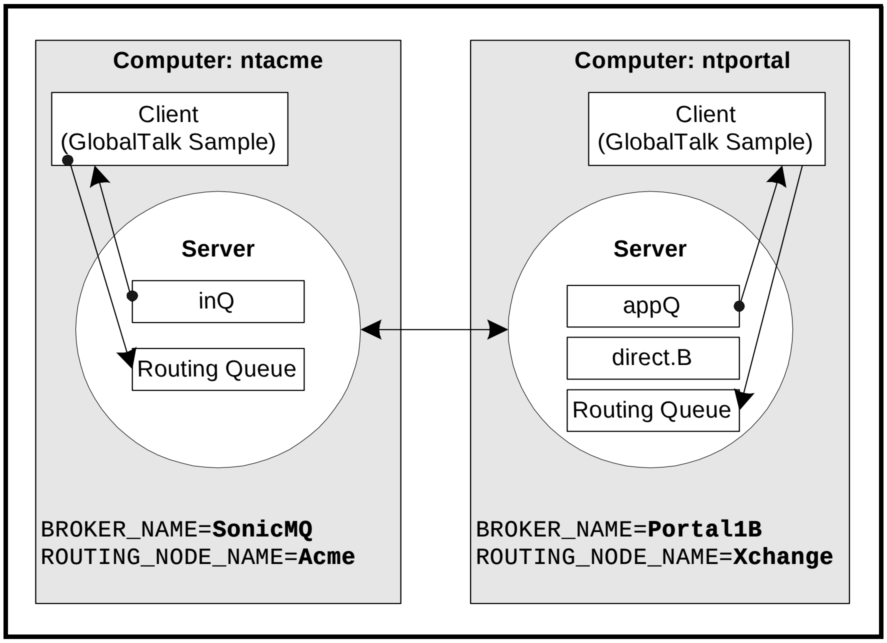


Figure 38. Configuration for Dynamic Routing Architecture

Before You Start

The Admin shell scripts contain the commands to set up the security, routing, and queues for the portal and trading partner used in this example. When you set up your Portal and Trading Partner, you will modify these scripts to include the names of your Portal and Trading Partner machines.

The associated scripts for the Admin Shell management tool are contained in `C:\Sonic\samples\Marketplace\scripts` and are printed at the end of this chapter. If you do not find the scripts in the `\scripts` directory specified here, you can copy the scripts from this book into text files and store them in the directory path used in this chapter.

Before you continue, perform the following steps to determine your Portal and Trading Partner machine names and set the `admin.echo` system property.

Note Perform the following procedures on **BOTH** the Portal and Trading Partner machines.

Determining Your Machine Names

On the Portal machine and on the Trading Partner machine, perform the following steps to determine your machine names:

1. From the Windows desktop select `Start >> Settings >> Control Panel`.
2. Double-click the **Network** icon.
3. Select the **Identification** tab.
4. Record your Computer Names:
   - Portal (to use in place of **ntportal)**:___________________
   - Trading Partner (to use in place of **ntacme)**:___________________

Installing SonicMQ for Your Portal and Trading Partner

Perform the following steps to install SonicMQ on your Portal and Trading Partner machines.

Note Remember to use the appropriate name for the Portal machine in your Trading Partner setup.

➤ To install SonicMQ on your Portal machine:

1. Install SonicMQ to the `c:\sonic` directory on the Portal machine **ntportal**.

   Change the name of the server to **PortalB** during the installation.

Note This server name change is not strictly necessary unless you plan to create a cluster in the future with this server.

2. Check your SonicMQ installation. Select:

   ```
   Start >> Programs >> Progress SonicMQ >> Start Broker
   ```

   Ensure that the server starts. You might consider opening the SonicMQ Explorer to further check your installation.

   Shut down the server after you confirm your installation.

➤ To install SonicMQ on your Trading Partner machine:

1. Install SonicMQ to the `c:\sonic` directory on the Trading Partner machine **ntacme**.
2. Check your SonicMQ installation. Select:

   ```
   Start >> Programs >> Progress SonicMQ >> Start Broker
   ```

   Ensure that the server starts. You might consider opening the SonicMQ Explorer to further check your installation.

   Shut down the server after you confirm your installation.

Setting the admin.echo System Property

This example assumes you are using a Windows system and uses the Admin shell script feature that echoes commands to the console. If you are using a UNIX or Linux system, you might consider modifying your system behavior accordingly.

It is useful to edit the `admin.bat` file to set the system property for `admin.echo=true`.

➤ To set the system property for admin.echo:

1. Edit `c:\sonic\bin\Admin.bat.`
2. Find the line that begins:

   ```
   “%SONICMQ_JRE%” %SONICMQ_SSL_CLIENT% ...
   ```
3. Add the following parameter to the command that starts Admin:

   ```
   -Dadmin.echo=true
   ```

   The line should now read as follows:

   ```
   “%SONICMQ_JRE%” %SONICMQ_SSL_CLIENT% -Dadmin.echo=true -cp...
   ```

Setting Up the Portal: Xchange

Perform the following procedures on your Portal machine (**ntportal**) to set up a Portal.

Note Remember to use the appropriate name for the Portal machine in your setup.

➤ **To preconfigure the Portal Admin shell script:**

The Admin shell scripts contain the commands to set up the security, routing, and queues for the Portal and Trading Partner used in this example. You must preconfigure the `Portal_Add_TP.txt` Admin shell script with the correct URL for the SonicMQ server. Change the following parameter based on your setup, using the appropriate name for your Trading Partner machine (see the "Determining Your Machine Names" section).

1. Replace **ntacme** with your machine name in the following URL:

   ACME_URL → `tcp://ntacme:2506`

   Your Trading Partner URL: tcp://____________________:2506

2. Edit the file `Portal_Add_TP.txt in the directory c:\sonic\samples\Marketplace\scripts`

   Replace **ACME_URL** with the URL for your trading partner machine in the following line of the `Portal_Add_TP.txt` script:

   ```
   set routing Acme "ACME_URL" thePortal pwd 3600 static
   ```

➤ **To set up your Portal for security and global routing:**

1. Edit `c:\sonic\broker.ini` on the portal machine, changing the following properties as shown:

   ```
   ENABLE_SECURITY=TRUE
   ROUTING_NODE_NAME=Xchange
   ```

2. Re-create the database using the SonicMQ dbtool to reconfigure the basic tables and create the security database. Open a console window and enter the following commands in the `\sonic\bin` directory:

   ```
   c:\sonic\bin> dbtool /r basic
   c:\sonic\bin> dbtool /c security
   ```

3. Start the server:

```
Start >> Programs >> Progress SonicMQ >> Start Broker
```

➤ **To use scripts to configure security, routing, and queues:**

1. On the Portal machine, open a console window and go to the install directory, `c:\sonic`.

2. Enter the following commands to pipe in each of the following Admin shell scripts from the `\JumpStart\Scripts` directory:

```
c:\Sonic> bin\Admin <
samples\Marketplace\scripts\Portal_Config_Setup.txt

c:\Sonic> bin\Admin <
samples\Marketplace\scripts\Portal_Add_TP.txt

c:\Sonic> bin\Admin <
samples\Marketplace\scripts\Portal_Broker_Setup.txt
```

Note If you have clusters, you must run the `Portal_Broker_Setup.txt` step on *each* of the servers in the cluster. Run the other two scripts on the configuration server only. You should also give the *`direct.*`* queue a unique name (direct.A, direct.B, etc.) for each server prior to setting up.

Setting Up the Trading Partner: Acme

Perform the following procedures to set up a Trading Partner.

Note Remember to use the appropriate name for the Trading Partner machine in your setup.

➤ To preconfigure the Trading Partner AdminShell script:

The Admin Shell scripts contain the commands to set up the security, routing, and queues for the portal and trading partner used in this example. You must preconfigure the `TP_Setup.txt` Admin Shell script with the correct URL for the SonicMQ server. Change the following parameters based on your setup, using the appropriate name for the portal machine (see the “Determining Your Machine Names” section).

1. Replace **ntportal** with your machine name in the following URLs:

   PORTAL_URL → `tcp://ntportal:2506`

   Your Portal URL: tcp://____________________:2506

2. Edit the file `TP_Setup.txt in the directory c:\sonic\samples\Marketplace\scripts.`

   Replace **PORTAL_URL** with the URL for your machine in the following lines of the `TP_Setup.txt` script:

```
// Portal is at "PORTAL_URL". Use Load-balanced connection.
// Advertising is required connection to clusters.
set routing Xchange "PORTAL_URL" AcmeCo pwd 3600 lb advertise
```

➤ To set up your Trading Partner for security and global routing:

1. Edit the `c:\sonic\broker.ini` file on the trading partner machine, changing the following properties as shown:

```
ENABLE_SECURITY=TRUE
ROUTING_NODE_NAME=Acme
```

2. Re-create the database using the SonicMQ dbtool to reconfigure the basic tables and create the security database. Open a console window and enter the following commands in the `c:\sonic\bin` directory:

   ```
   c:\sonic\bin> dbtool /r basic
   c:\sonic\bin> dbtool /c security
   ```

3. Start the broker:

   ```
   Start >> Programs >> Progress SonicMQ >> Start Broker
   ```

➤ **To use scripts to configure the security, routing, and queues:**

1. On the trading partner machine, open a console window and go to the install directory:

   ```
   c:\sonic
   ```

2. Enter the following command to pipe in the Admin Shell scripts from the `\JumpStart\Scripts` directory:

   ```
   c:\sonic> bin\Admin < samples\Marketplace\scripts\TP_Setup.txt
   ```

Testing Your Setup with the GlobalTalk Sample Application

You are now ready to test your setup using a sample application. If you wish, you can first start the SonicMQ Explorer on one or both of the machines and look at the Queues, Users, Groups, and Routings:

```
Start >> Programs >> Progress SonicMQ >> Explorer
```

In the SonicMQ Explorer, you can connect with a user name of *Administrator* and the default password of *Administrator*. Once you connect, you can review users, groups, routings, and queues to investigate the scripts.

The following steps tell you how to send messages between the trading partner and portal using the `GlobalTalk` application. See the "The GlobalTalk Application (PTP)" section for more information about this application.

➤ To run GlobalTalk on each server with the appropriate startup options:

1. On the trading partner **Acme,** simulate a trading partner application that has logged in as `user1/pwd`. This application will send to the portal at `Xchange::appQ` and listen on the `inQ`. Open a console window and enter the following command in the `GlobalTalk` directory:

   ```
   cd c:\sonic\samples\QueuePTP\GlobalTalk
   ```
   ```
   ..\..\SonicMQ GlobalTalk -u user1 -p pwd -qs Xchange::appQ -qr
   inQ
   ```

2. On the portal **Xchange,** simulate a routing application that has logged in as `pUser/pwd`. This application will send to the partner at `Acme::inQ` and listen on the `appQ`. Open a console window and enter the following command in the `GlobalTalk` directory:

   ```
   cd c:\sonic\samples\QueuePTP\GlobalTalk
   ```
   ```
   ..\..\SonicMQ GlobalTalk -u pUser -p pwd -qs Acme::inQ -qr appQ
   ```

3. You can now type messages in either `GlobalTalk` application and have them sent to the remote queue on the other routing node.

Troubleshooting Your Setup

The following issue might arise when you use the JumpStart scripts.

Permission Problems When Sending Messages to Valid Queues

You might encounter permission problems when you send messages to queues that you know to be valid. This problem might be caused by a security setting in the security startup scripts.

The JumpStart scripts are modeled on portal/marketplace situations where security is an issue. There is a command line in the setup scripts for both the Portal and the Trading Partner that sets the default security to deny all users access to all queues. This line is:

```
set queue acl # PUBLIC -snd -rcv
```

You can remove this line from the security setup scripts: `Portal_Config_Setup.txt` and `TP_Setup.txt`.

Sample Application and Scripts

This chapter uses the `GlobalTalk` application and four Admin Shell scripts. The following sections provide more information about `GlobalTalk` and the scripts.

The GlobalTalk Application (PTP)

This example uses the `GlobalTalk` application to illustrate your trading partner and portal setup. This section explains the `GlobalTalk` application. The steps to start `GlobalTalk` and send messages between the Accounting and Orders windows are provided here to illustrate how the application might be used. You do not need to perform these steps to test your trading partner-portal setup.

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

➤ To start GlobalTalk:

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

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

➤ Talking:

☐ In the **`Orders`** window, type any text and then press **Enter**.
The text is displayed in only the **`Accounting`** window.

In the **`Accounting`** window, type text and then press **Enter**.
The text is displayed in only the **`Orders`** window.

The Admin Shell Scripts

The following four scripts set up your Portal and Trading Partner configurations for the example used in this chapter.

Shortcuts used in this example include:

- Using default acceptors (tcp on port 2506)
- Setting up all routing nodes as stand-alone servers (not clusters)

To set up your Portal, use these scripts:

- **Portal_Broker_Setup.txt** — Sets up queues for the servers
- **Portal_Config_Setup.txt** — Sets up Portal security
- **Portal_Add_TP.txt** — Adds a new user to the Portal and sets up routing to the Trading Partner

To set up your Trading Partner, use this script:

- **TP_Setup.txt** — Adds new users, sets up queues, and sets up security for the Trading Partner

Portal_Broker_Setup

The Portal_Broker_Setup.txt script provides a setup file for each server in the Portal cluster. This script:

- Creates incoming global queues
- Overrides the properties of the system queues

```
//
// Setup file for EACH server in the portal cluster.
//
connect broker localhost Administrator Administrator

// Create incoming global queues.
// -- appQ will exist on all servers
// -- direct.B will only exist on one (e.g. PortalB)
set queue appQ global
set queue direct.B global

// Override properties of system queues.
set queue SonicMQ.deadMessage local 1200,1400,20000
set queue SonicMQ.routingQueue local 1200,1400,4000

// Close the Admin Shell.
bye
```

Portal_Config_Setup

The Portal_Config_Setup.txt script provides a setup file for the Portal security database. This script:

- Creates a group for all the internal portal applications
- Creates a group for all the Trading Partners
- Sets the Access Control List (ACL) to:
  - Prevent everyone from accessing all Queues
  - Allow Portal Applications to receive from appQ/direct queues
  - Allow TradingPartners to send only to appQ/direct queues

```
//
// Setup file for Portal security database
//
connect broker localhost Administrator Administrator

// Create a group for all the internal portal applications.
// Add the necessary local users to this group
add group RoutingUsers
add user pUser pwd
add groupuser RoutingUsers pUser

// Create a group for all the trading partners
// (routing users will be added to this group, later).
add group TradingPartners

// ACLs -- Prevent everyone from accessing all Queues
// unless explicitly granted below.
set queue acl # PUBLIC -snd -rcv

// ACLs -- Portal Applications can receive from appQ/direct queues
// and send to the trading partner's inQ.
set queue acl appQ     RoutingUsers -snd +rcv
set queue acl direct.* RoutingUsers -snd +rcv
set queue acl inQ      RoutingUsers +snd -rcv

// ACLs -- the TradingPartners can only send to appQ/direct queues.
set queue acl appQ     TradingPartners +snd -rcv
set queue acl direct.* TradingPartners +snd -rcv

// Close the Admin Shell.
bye
```

Portal_Add_TP

The Portal_Add_TP.txt script provides a setup file to add a new user to the Portal and to set up routing to the Trading Partner. This script:

- Creates a new user in the Trading Partners group
- Sets up the routing to the Trading Partner

```
//
// Setup file for adding user "Acme" to the configuration
// at the Portal (Xchange).
//
connect broker localhost Administrator Administrator

// Create the new user in the TradingPartners group.
add routing user AcmeCo Acme pwd
add groupuser TradingPartners AcmeCo

// Set up the routing to Acme.
// This is not a load-balanced connection. Always use
// this routing (static) and not any advertised routings
// from the partner.
set routing Acme "ACME_URL" thePortal pwd 3600 static

// Close the Admin Shell.
bye
```

TP_Setup

The TP_Setup.txt script provides a setup file for the Trading Partner. This script:

- Creates local application users in an Applications group
- Sets the Portal to PORTAL_URL using a load-balanced connection
- Configures the Portal user to allow Xchange to call back into this routing node
- Creates incoming global queues
- Overrides properties to the system queues
- Sets the Access Control List (ACL) to:
  - Prevent everyone from accessing all Queues
  - Allow applications to send to appQ and direct queues
  - Allow the portal user to send only to inQ

```
//
// Setup file for Trading Partner (Acme)
//
connect broker localhost Administrator Administrator

// Create local application users (in an "Applications" group)
// Add sample users to the group
add group Applications
add user user1 pwd
add user user2 pwd
add user user3 pwd
add groupuser Applications user1
add groupuser Applications user2
add groupuser Applications user3

// Portal is at "PORTAL_URL". Use Load-balanced connection.
// Advertising is required connection to clusters.
set routing Xchange "PORTAL_URL" AcmeCo pwd 3600 lb advertise
```

```
// Configure thePortal user (to allow "Xchange" to
// call back into this routing node)
add routing user thePortal "Xchange" pwd

// Create incoming global queues.
set queue inQ global

// Override properties to the system queues.
set queue SonicMQ.deadMessage local 1200,1400,15000
set queue SonicMQ.routingQueue local 1200,1400,2000

// ACLs -- Prevent everyone from accessing all queues
// (unless explicitly granted below).
set queue acl # PUBLIC -snd -rcv

// ACLs -- Applications can send to appQ and direct queues
// (on Xchange), and receive from the inQ (defined above).
set queue acl appQ     Applications +snd -rcv
set queue acl direct.* Applications +snd -rcv
set queue acl inQ      Applications -snd +rcv

// ACLs -- the portal routing user can only send to inQ.
set queue acl inQ    thePortal    +snd -rcv

// Close the Admin Shell.
bye
```

Appendix A

Performance Tuning

Your SonicMQ application performance will vary based on your specific functional requirements and your individual deployment environment. However, you can tune some parameters of your SonicMQ configuration to optimize the overall performance of your implementation. Depending on your application, you might choose to adjust some or all of the parameters discussed in the following sections.

Tuning Your JVM Properties

Your choice of Java Virtual Machine (JVM), Java heap size, and memory settings will have significant impact on your SonicMQ performance. The following sections discuss some issues you should consider when optimizing your JVM.

Choosing a Java Virtual Machine for the SonicMQ Server

Both the SonicMQ server and standard client are written in Java. The Java Virtual Machine that you use to run the SonicMQ broker can have a significant impact on overall messaging performance. Recent JVM advances allow for just-in-time compilation of Java classes, enhanced garbage collection, efficient input and output processing, and other significant capabilities. These advances can improve overall performance by as much as 300%, making the choice of JVM critical to attaining high performance levels.

Setting the Java Heap Size

A significant performance factor is the size of the Java heap, which you can specify in the JVM command line with the `-mx` parameter as explained in "Tuning JVM Parameters." Typically this parameter is set to 128 or 256MB. If you plan to send or receive very large messages or have multiple concurrent sessions in your application, you should increase the java memory for the client machine.

You should determine your maximum heap size based on your available memory and on the size and number of messages and queues you anticipate handling. The following sections discuss these considerations.

Using the Maximum Available Memory for the Server

To optimize performance, you should set the maximum Java heap size possible for your SonicMQ broker. This maximum size should correspond to the available memory on your machine. The more memory you set, the less Java will use the garbage cleanup. Less use of the garbage cleanup results in better performance.

However, you should be careful not to set the Java heap size too high. If this parameter exceeds the memory available to the JVM process, performance can significantly degrade as a result of page swapping in the underlying operating system. The memory available to the JVM might not match the total memory in the server machine due to the memory requirements of other processes. In this case, lowering the total heap for the JVM will increase performance.

Anticipating the Size and Number of Messages and Queues on the Server

You should base your JVM memory size on the size and number of messages you expect to have in all queues on your SonicMQ server. Similarly, you should set the size of your queues to the maximum size of the messages you expect to store. You can determine the required JVM memory size by anticipating the maximum message size you expect to store in the queue, and multiplying that size by the number of messages you plan to store at one time on the queue. Total the memory needed for all queues in your application, and base your JMV memory size on this figure.

Tuning JVM Parameters

Table 10 describes properties in the broker.ini and the startbr.bat (NT) and setenv (UNIX) files that affect performance for the JVM and the SonicMQ server.

Table 10. JVM Settings

Option	*Description*
-ms<n>	Sets the initial Java heap size
-mx<n>	Sets the maximum Java heap size
-oss<n>	Sets the maximum Java stack size for any thread
-ss<n>	Sets the maximum native stack size for any thread

Table 11 lists the default settings for the JVM_PARAM and MAX_LOG_FILE_SIZE parameters. These parameters are set during the SonicMQ installation based on your product choice. The MAX_LOG_FILE_SIZE resides in `startbr.bat` on Windows. On UNIX, use `setenv`.

Table 11. JVM Settings for SonicMQ Editions

SonicMQ Edition	*Suggested Settings*

SonicMQ Developer Edition	JVMParamString=”-mx32m -ss64k -oss64k”
	logFileParamString=”10000000”; // 10 MB

SonicMQ Professional Developer Edition	JVMParamString=”-ms32m -mx256m -oss64k”
	logFileParamString=”104857600”; // 100 MB

SonicMQ E-Business Edition	JVMParamString=”-ms32m -mx256m -oss64k”
	logFileParamString=”104857600”; // 100 MB

Setting Buffer Limits in Message Flow Control

When clients send messages at a faster rate than they can be received at their destination, a server must save them for delivery. When the server's capacity limits are reached, the sending client must be throttled using flow control to avoid losing messages. This throttling results in the reduction of the client's send rate. You can either predefine the server's capacity limit administratively or determine this limit based on limitations in memory or disk space. How and when you apply flow control can significantly alter the performance results of your messaging system.

When you configure a server with a high buffer limit, messages accumulate in that server's memory before flow control is applied. As a result, the sending client attains a high level of performance until the flow control point is reached.

On the other hand, if you configure the server with a lower buffer limit, there is a shorter period of time during which client send rates are higher than receive rates. As a result, less memory is used on messages buffered for delivery.

Another effect of setting high buffer size limits is increased delivery time for each message, since buffered messages spend more time in memory and take longer to arrive at a receiver.

In addition to send rates, you should consider the total number of messages delivered in a system under load. Excessive buffering typically hinders absolute throughput. Flow control takes effect any time the send rate exceeds the receive rate. As a result, messages might be buffered in the server and remain undelivered for a measurable period of time.

The appropriate size limits that govern flow control will vary between applications. You might find it advantageous to enable a high send rate for a client, particularly if the number of messages will be small and buffering will not have a great effect.

SonicMQ allows you to adjust the effect of flow control by setting tunable buffer sizes for Pub/Sub and Queues. For queue-based messaging, the size of the queue affects when flow control is applied:

- You can set the parameter OUTPUT_QUEUE_SIZE in the `broker.ini` file to adjust the buffer size for each client's delivery queue. The default value of this parameter is 150000 bytes.

- You can set the parameter GUAR_QUEUE_SIZE to set the buffer size limit per client for messages that have been delivered and are waiting for acknowledgement. The default value of this parameter is 150000 bytes.

Setting Queue Save/Retrieve Extents

When flow control is not appropriate, it is possible to have a very large queue by forcing some messages to disk rather than being stored in memory. For these cases SonicMQ provides the Save_Extent parameter on every queue. The save/retrieve extents define:

- The maximum size of the queue (max extent)
- The in-memory portion of the queue (save extent)
- When to retrieve messages from the database (retrieve extent)

The Save_Extent parameter defines the queue size at which messages are saved to the database. When the in-memory portion of the queue falls below the retrieve extent, messages are retrieved from the database to fill the queue. By setting the save extent larger than the max size, you can ensure that queue messages are never saved to the databases. This technique avoids the overhead of database operations and is appropriate for fast-moving queues.

Reducing the Number of Syncpoints

A **syncpoint** in SonicMQ is the time when the running state of the message server is saved in the recovery log files. This information ensures the delivery of persistent messages. (Non-persistent messages are not written to the log.) Syncpoints provide a safe starting point for recovery operations in the case of server machine failure and allow older recovery information to be discarded once the syncpoint is complete.

In SonicMQ, a syncpoint is performed when the server fills one log file and switches to the second. The length of the log files therefore determines how often the syncpoints occur. You set the log file size with the parameter MAX_LOG_FILE_SIZE. See Table 11 for the default settings for the MAX_LOG_FILE_SIZE parameter.

You can also set the SYNCPOINT_INTERVAL parameter to determine the number of bytes of log between syncpoints. You should base the length of this parameter on the anticipated sizes of your messages.

Because the syncpoint process consumes resources in the server, longer files yield higher performance levels overall. The SonicMQ server provides a warning when syncpoint operations account for more than 50% of the total log file size.

Choosing Automatic Message Acknowledgement

When you use messaging, you can choose to have messages acknowledged in one of the following ways:

- Acknowledged automatically by message receivers
- Acknowledged through *client acknowledge*, which is under the control of the receiver

When you choose CLIENT_ACKNOWLEDGEMENT, the server cannot send subsequent messages until the acknowledgment occurs. Choosing to have messages acknowledged asynchronously can help your system avoid unnecessary slowdowns when using guaranteed messages.The DUPS_OK_ACKNOWLEDGE is the fasted acknowledgement mode you can choose.

Disk Drive Caching

Warning ***DO NOT*** use disks configured to use a write cache. This can lead to failure to recover messages in the event of a server failure. Reliability cannot be guaranteed if you use disks configured to buffer writes.

Disk file access from the server can have a major influence on overall performance. Increased drive speeds directly translate to higher message throughput when your system processes guaranteed messages. Many disk drive controllers support write caches that allow disk writes to be delayed, increasing write speeds for the operating system. However, while a write cache increases performance, it also increases the possibility that messages will be lost when a server machine fails. For this reason you should not use disks configured to use a write cache.

Note Mapped drives typically cache disk writes.

Using Queue Prefetch

SonicMQ supports prefetching messages from a queue to optimize overall throughput. Prefetching allows a client to receive messages from the SonicMQ server before the client explicitly requests the messages, eliminating the overhead of server requests on a per-message basis. However, prefetching also changes the operation of the SonicMQ system by allowing messages to accumulate at the client until the number of messages reaches the application-defined count.

You can achieve some performance gain with prefetching primarily on lightly loaded servers, where a receiving client tends to govern overall throughput. When the server is operating at full capacity, other factors (such as queue size and disk I/O) tend to limit message-delivery rates.

Queue Cleanup Thread

The queue cleanup thread parameters specify how often SonicMQ checks for expired messages. This checking takes time, and so reducing the frequency of the checks, or eliminating them altogether, can help improve your SonicMQ performance.

You can set the following `broker.ini` parameters to adjust the frequency of or eliminate the queue cleanups:

- ENABLE_DYNAMIC_QUEUE_CLEANUP — This parameter defaults to `TRUE`, enabling queue cleanup. Setting this parameter to `FALSE` eliminates queue cleanup.
- QUEUE_CLEANUP_INTERVAL — This parameter determines the amount of time between cleanup (in seconds).

Message Size

As you increase message size, you should adjust other parameters to optimize your system performance with your selected message size. These parameters include message latency, message type, and log queue size.

Message Type

You might be able to reduce your message size by converting text messages to bytes messages. Each character in a text message is two bytes, which doubles the size of the message (compared to the size of the same message converted to bytes). Unless your application requires double byte size, you can reduce your text message size by converting to bytes.

Latency

The output queue size directly affects the message latency. As a large number of messages collects in a queue, the messages coming into the queue take longer to be delivered. To optimize your performance, you should base your queue size on the size of the messages, so that the number of messages buffered at any one time is minimized.

Log Queue Size

The log queue size determines the maximum amount of memory (in bytes) that can be used by messages being written to the log. If this parameter is set too low for your application, large messages (that is, messages whose size exceed the log queue size) will be logged one by one. This individual logging decreases your system performance. Setting your log queue size to a higher value allows more events to be queued up in memory before the log is flushed to disk. If the flush operation takes a significant amount of time and there are a significant number of publishers/senders, flow control might be activated to reduce publish/subscribe rates until space opens up on the log's event queue.

You can set the LOG_QUEUE_SIZE parameter in the `broker.ini` file. The default value of this parameter is 500000 bytes.

Security

Security requires that your system use encryption, which results in slower performance. If only some of your clients require security, you can increase performance by enabling security only for those clients that require it.

You can enable or disable security by editing the parameter ENABLE_SECURITY in the `broker.ini` file. The default setting for this parameter is `TRUE`, which enables security. Changing this value to `FALSE` disables security.

Index