



# Sun Java System Message Queue 4.1 Technical Overview



Sun Microsystems, Inc.  
4150 Network Circle  
Santa Clara, CA 95054  
U.S.A.

Part No: 819-7759  
September 2007

Copyright 2007 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights – Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, the Solaris logo, the Java Coffee Cup logo, docs.sun.com, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Products covered by and information contained in this publication are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical or biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

---

Copyright 2007 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs brevets américains ou des applications de brevet en attente aux États-Unis et dans d'autres pays.

Cette distribution peut comprendre des composants développés par des tierces personnes.

Certains composants de ce produit peuvent être dérivés du logiciel Berkeley BSD, licenciés par l'Université de Californie. UNIX est une marque déposée aux États-Unis et dans d'autres pays; elle est licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, le logo Solaris, le logo Java Coffee Cup, docs.sun.com, Java et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux États-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux États-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui, en outre, se conforment aux licences écrites de Sun.

Les produits qui font l'objet de cette publication et les informations qu'il contient sont régis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes chimiques ou biologiques ou pour le nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.

# Contents

---

<b>Preface</b> .....	7
<b>1 Messaging Systems: An Introduction</b> .....	15
Message-Oriented Middleware (MOM) .....	15
JMS as a MOM Standard .....	20
JMS Messaging Objects and Patterns .....	21
Administered Objects .....	22
▼ To Use Administered Objects as Destinations .....	23
Message Queue: Elements and Features .....	24
The Message Queue Service .....	24
Connecting to the Broker .....	25
The Broker .....	26
Client Runtime Support .....	27
Administration .....	28
Scaling the Message Queue Service .....	28
Message Queue as an Enabling Technology .....	30
Message Queue Feature Summary .....	30
<b>2 Client Programming Model</b> .....	31
Design and Performance .....	31
Messaging Domains .....	32
Point-To-Point Messaging .....	32
Publish/Subscribe Messaging .....	34
Domain-Specific and Unified APIs .....	36
Programming Objects .....	37
Connection Factories and Connections .....	38
Sessions .....	39

Messages .....	40
Producing a Message .....	42
Consuming a Message .....	43
Synchronous and Asynchronous Consumers .....	43
Using Selectors to Filter Messages .....	43
Using Durable Subscribers .....	44
The Request-Reply Pattern .....	44
Reliable Messaging .....	46
Acknowledgements .....	46
Transactions .....	47
Persistent Storage .....	48
A Message's Journey Through the System .....	48
Message Production .....	49
Message Handling and Routing .....	50
Message Consumption .....	50
Message End-of-Life .....	50
Working with SOAP Messages .....	51
▼ To Get Reliable SOAP Messaging .....	51
Java and C Clients .....	52
<b>3 Message Queue Service .....</b>	<b>53</b>
Component Services .....	53
Connection Services .....	55
Destinations and Routing Services .....	56
Persistence Services .....	59
Security Services .....	60
Monitoring Services .....	63
Administration Tools and Tasks .....	65
Administration Tools .....	65
JMX-Based Administration .....	67
Supporting a Development Environment .....	68
Supporting a Production Environment .....	68
Scaling Messaging Operations .....	70

---

<b>4</b>	<b>Broker Clusters</b> .....	71
	Cluster Models .....	71
	Conventional Clusters .....	72
	High Availability Clusters .....	73
	Message Delivery .....	74
	Destination Attributes .....	75
	Clustering and Destinations .....	76
	Cluster Configuration .....	80
	Cluster Synchronization .....	81
	Choosing a Clustering Model .....	82
<b>5</b>	<b>Message Queue and J2EE</b> .....	83
	JMS/J2EE Programming: Message-Driven Beans .....	83
	J2EE Application Server Support .....	85
	JMS Resource Adapter .....	85
<b>A</b>	<b>Message Queue Implementation of Optional JMS Functionality</b> .....	87
	Optional Features .....	87
<b>B</b>	<b>Message Queue Features</b> .....	89
	Feature List .....	91
	<b>Glossary</b> .....	109
	<b>Index</b> .....	113



# Preface

---

This book, the *Sun Java System Message Queue 3.7 URI Technical Overview*, provides an introduction to the technology, concepts, architecture, capabilities, and features of the Message Queue messaging service.

As such, this book provides the foundation for other books within the Message Queue documentation set, and should be read first.

## Who Should Use This Book

This guide is meant for application developers, administrators, and other parties who plan to use the Message Queue product or who wish to understand the technology, concepts, architecture, capabilities, and features of the product.

- An application developer is responsible for writing Message Queue client applications that use the Message Queue service to exchange messages with other client applications.
- An administrator is responsible for setting up and managing a Message Queue messaging service. This book does not assume any knowledge or understanding of messaging systems.

This book does not assume any knowledge of the Java Message Service (JMS) specification, which is implemented by the Message Queue service.

## Before You Read This Book

There are no prerequisites to this book. You should read this book to gain an understanding of basic Message Queue concepts before reading the Message Queue developer and administration guides.

## How This Book Is Organized

This guide is designed to be read from beginning to end; each chapter builds on information contained in earlier chapters. The following table briefly describes the contents of each chapter.

TABLE P-1 Book Contents and Organization

Chapter	Description
Chapter 1, “Messaging Systems: An Introduction,”	Introduces messaging middleware technology, discusses the JMS standard, and describes the Message Queue service implementation of that standard.
Chapter 2, “Client Programming Model,”	Describes the JMS programming model and how you can use the Message Queue client runtime to create JMS clients. Describes runtime support for C++ clients and for the transport of SOAP messages.
Chapter 3, “Message Queue Service,”	Discusses administrative tasks and tools and describes broker services used to configure connections, routing, persistence, security, and monitoring.
Chapter 4, “Broker Clusters,”	Discusses the architecture and use of Message Queue broker clusters.
Chapter 5, “Message Queue and J2EE,”	Explores the ramifications of implementing JMS support in a J2EE platform environment
Appendix A, “Message Queue Implementation of Optional JMS Functionality”	Describes how the Message Queue product handles JMS optional items
Appendix B, “Message Queue Features”	Lists Message Queue features, summarizes steps needed to implement these, and provides reference for further information
Glossary	Provides information about terms and concepts you might encounter while using Message Queue

## Related Documentation

The documents that comprise the Message Queue documentation set are listed in the following table in the order in which you would normally use them.



TABLE P-2 Message Queue Documentation Set

Document	Audience	Description
<i>Sun Java System Message Queue 4.1 Installation Guide</i>	Developers and administrators	Explains how to install Message Queue software on Solaris, Linux, and Windows platforms.
<i>Sun Java System Message Queue 4.1 Release Notes</i>	Developers and administrators	Includes descriptions of new features, limitations, and known bugs, as well as technical notes.
<i>Sun Java System Message Queue 4.1 Technical Overview</i>	Developers and administrators	Describes Message Queue concepts, features, and components.
<i>Sun Java System Message Queue 4.1 Developer's Guide for Java Clients</i>	Developers	Provides a quick-start tutorial and programming information for developers of Java client programs using the Message Queue implementation of the JMS or SOAP/JAXM APIs.
<i>Sun Java System Message Queue 4.1 Administration Guide</i>	Administrators, also recommended for developers	Provides background and information needed to perform administration tasks using Message Queue administration tools.
<i>Sun Java System Message Queue 4.1 Developer's Guide for C Clients</i>	Developers	Provides programming and reference documentation for developers of C client programs using the Message Queue C implementation of the JMS API (C-API).

## Online Help

Message Queue 4.1 includes command-line utilities for performing Message Queue message service administration tasks.

Message Queue 4.1 also includes a graphical user interface (GUI) administration tool, the Administration Console (`imqadmin`). Context-sensitive help is included in the Administration Console; see “Administration Console Online Help” in *Sun Java System Message Queue 4.1 Administration Guide*.

## JavaDoc

JMS and Message Queue API documentation in JavaDoc format is provided at the following location:

Platform	Location
Solaris	/usr/share/javadoc/imq/index.html
Linux	/opt/sun/mq/javadoc/index.html
Windows	IMQ_HOME/javadoc/index.html

This documentation can be viewed in any HTML browser. It includes standard JMS API documentation, as well as Message Queue-specific APIs for Message Queue administered objects, which are of value to developers of messaging applications.

## Example Client Applications

Message Queue provides a number of example client applications to assist developers.

### Example Java Client Applications

Example Java client applications are located in the following directories, depending on platform. See the README file located in these directories and in each of their subdirectories.

Platform	Location
Solaris	/usr/demo/imq/
Linux	/opt/sun/mq/examples
Windows	IMQ_HOME/demo/

### Example C Client Programs

Example C client applications are located in the following directories, depending on platform. See the README file located in these directories.

Platform	Location
Solaris	/opt/SUNWimq/demo/C/
Linux	/opt/sun/mq/examples/C/
Windows	IMQ_HOME/demo/C/

# The Java Message Service (JMS) Specification

The JMS specification can be found at the following location:

(<http://java.sun.com/products/jms/docs.html>)

The specification includes sample client code.

## Directory Variable Conventions

Message Queue makes use of three directory variables; how they are set varies from platform to platform. Table P-3 describes these variables and how they are used on the Solaris, Linux, and Windows platforms.

---

**Note** – The information in Table P-3 applies only to the standalone installation of Message Queue. When Message Queue is installed and run as part of an Application Server installation, the values of the directory variables are set differently: `IMQ_HOME` is set to `appServer_install_dir/imq` (where `appServer_install_dir` is the Application Server installation directory), and `IMQ_VARHOME` is set to `appServer_domainName_dir/imq` (where `appServer_domainName_dir` is the domain directory for the domain starting the Message Queue broker).

---

TABLE P-3 Directory Variable Conventions

Variable	Description
<b>IMQ_HOME</b>	<p>Used in Message Queue documentation to refer to the Message Queue base directory (root installation directory):</p> <ul style="list-style-type: none"> <li>■ On Solaris and Linux, there is no root Message Queue installation directory. Therefore <code>IMQ_HOME</code> is not used in Message Queue documentation to refer to file locations in Solaris and Linux.</li> <li>■ On Windows, the root Message Queue installation directory is set to the directory in which you unzip the Message Queue bundle.</li> </ul>
<b>IMQ_VARHOME</b>	<p>The <code>/var</code> directory in which Message Queue temporary or dynamically-created configuration and data files are stored. It can be set as an environment variable to point to any directory.</p> <ul style="list-style-type: none"> <li>■ On Solaris, <code>IMQ_VARHOME</code> defaults to the <code>/var/imq</code> directory.</li> <li>■ On Solaris, for Sun Java System Application Server, Evaluation Edition, <code>IMQ_VARHOME</code> defaults to the <code>IMQ_HOME/var</code> directory.</li> <li>■ On Linux, <code>IMQ_VARHOME</code> defaults to the <code>/var/opt/sun/mq</code> directory.</li> <li>■ On Windows, <code>IMQ_VARHOME</code> defaults to the <code>IMQ_HOME/var</code> directory.</li> </ul>

TABLE P-3 Directory Variable Conventions (Continued)

Variable	Description
<b>IMQ_JAVAHOME</b>	<p>An environment variable that points to the location of the Java runtime environment (JRE) required by Message Queue executables:</p> <ul style="list-style-type: none"> <li>■ On Solaris, <b>IMQ_JAVAHOME</b> looks for the latest JDK, but a user can optionally set the value to wherever the preferred JRE resides.</li> <li>■ On Linux, Message Queue first looks for the latest JDK, but a user can optionally set the value of <b>IMQ_JAVAHOME</b> to wherever the preferred JRE resides.</li> <li>■ On Windows, <b>IMQ_JAVAHOME</b> will be set to point to an existing Java runtime if a supported version is found on the system. If a supported version is not found, one will be installed.</li> </ul>

In this guide, **IMQ\_HOME**, **IMQ\_VARHOME**, and **IMQ\_JAVAHOME** are shown *without* platform-specific environment variable notation or syntax (for example, **\$IMQ\_HOME** on UNIX). Path names generally use UNIX directory separator notation (/).

## Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P-4 Typographic Conventions

Typeface	Meaning	Example
AaBbCc123	The names of commands, files, and directories, and onscreen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% you have mail.</code>
<b>AaBbCc123</b>	What you type, contrasted with onscreen computer output	<code>machine_name% <b>su</b></code> <code>Password:</code>
<i>aabbcc123</i>	Placeholder: replace with a real name or value	The command to remove a file is <i>rm filename</i> .
<i>AaBbCc123</i>	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the <i>User's Guide</i> . <i>A cache</i> is a copy that is stored locally. Do <i>not</i> save the file. <b>Note:</b> Some emphasized items appear bold online.

## Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, Korn shell, and Windows operating system.

TABLE P-5 Shell Prompts

Shell	Prompt
C shell	machine_name%
C shell for superuser	machine_name#
Bourne shell and Korn shell	\$
Bourne shell and Korn shell for superuser	#
Windows	C:\

## Symbol Conventions

The following table explains symbols that might be used in this book.

TABLE P-6 Symbol Conventions

Symbol	Description	Example	Meaning
[ ]	Contains optional arguments and command options.	ls [-l]	The -l option is not required.
{   }	Contains a set of choices for a required command option.	-d {y n}	The -d option requires that you use either the y argument or the n argument.
\${ }	Indicates a variable reference.	\${com.sun.javaRoot}	References the value of the com.sun.javaRoot variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

## Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (<http://www.sun.com/documentation/>)
- Support (<http://www.sun.com/support/>)
- Training (<http://www.sun.com/training/>)

## Searching Sun Product Documentation

Besides searching Sun product documentation from the [docs.sun.com](http://docs.sun.com) web site, you can use a search engine by typing the following syntax in the search field:

```
search-term site:docs.sun.com
```

For example, to search for “broker,” type the following:

```
broker site:docs.sun.com
```

To include other Sun web sites in your search (for example, [java.sun.com](http://java.sun.com), [www.sun.com](http://www.sun.com), and [developers.sun.com](http://developers.sun.com)), use “sun.com” in place of “docs.sun.com” in the search field.

## Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

---

**Note** – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

---

## Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to <http://docs.sun.com> and click Send Comments. In the online form, provide the full document title and part number. The part number is a 7-digit or 9-digit number that can be found on the book's title page or in the document's URL. For example, the part number of this book is 819-7759.

# Messaging Systems: An Introduction

---

Sun Java™ System Message Queue is a messaging middleware product that implements and extends the Java Message Service (JMS) standard. If this statement makes perfect and deep sense to you, you should start by reading the section “[Message Queue: Elements and Features](#)” on [page 24](#). Otherwise, you should begin at the beginning.

This chapter describes the messaging technology that underlies products like Message Queue and explains how Message Queue implements and extends the JMS specification, which standardizes this technology. It covers the following topics:

- “[Message-Oriented Middleware \(MOM\)](#)” on [page 15](#)
- “[JMS as a MOM Standard](#)” on [page 20](#)
- “[Message Queue: Elements and Features](#)” on [page 24](#)
- “[Scaling the Message Queue Service](#)” on [page 28](#)
- “[Message Queue as an Enabling Technology](#)” on [page 30](#)

## Message-Oriented Middleware (MOM)

Because businesses, institutions, and technologies change continually, the software systems that serve them must be able to accommodate such changes. Following a merger, the addition of a service, or the expansion of available services, a business can ill afford to recreate its information systems. It is at this most critical point that it needs to integrate new components or to scale existing ones as efficiently as possible. The easiest way to integrate heterogeneous components is not to recreate them as homogeneous elements but to provide a layer that allows them to communicate despite their differences. This layer, called *middleware*, allows software components (applications, enterprise java beans, servlets, and other components) that have been developed independently and that run on different networked platforms to interact with one another. It is when this interaction is possible that the network can become the computer.

As shown in [Figure 1–1](#), conceptually, middleware resides between the application layer and the platform layer (the operating system and underlying network services).

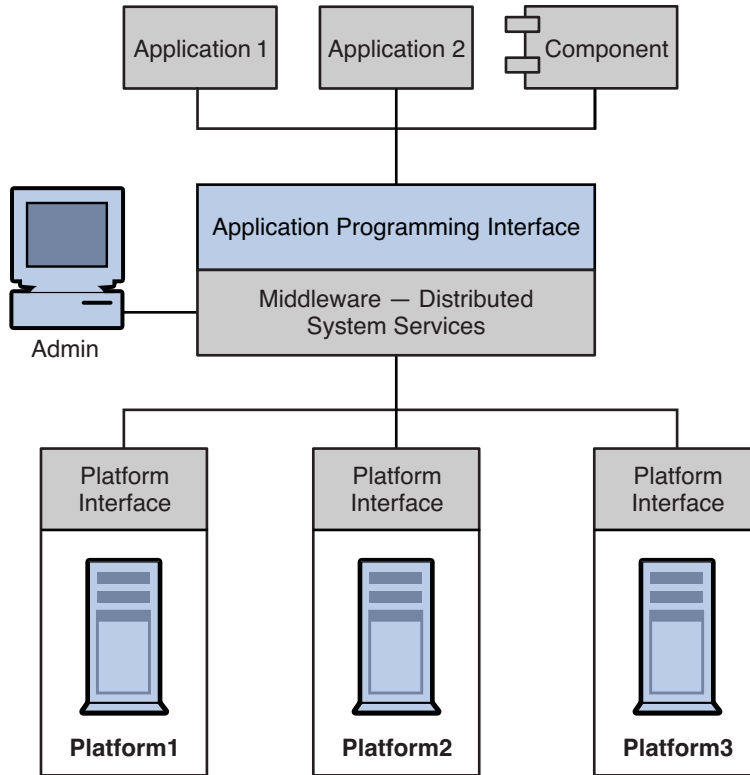


FIGURE 1-1 Middleware

Applications distributed on different network nodes use the application interface to communicate without having to be concerned with the details of the operating environments that host other applications nor with the services that connect them to these applications. In addition, by providing an administrative interface, this new, virtual system of interconnected applications can be made reliable and secure. Its performance can be measured and tuned, and it can be scaled without losing function.

Middleware can be grouped into the following categories:

- Remote Procedure Call or RPC-based middleware, which allows procedures in one application to call procedures in remote applications as if they were local calls. The middleware implements a linking mechanism that locates remote procedures and makes these transparently available to a caller. Traditionally, this type of middleware handled procedure-based programs; it now also includes object-based components.
- Object Request Broker or ORB-based middleware, which enables an application's objects to be distributed and shared across heterogeneous networks.



- Message Oriented Middleware or MOM-based middleware, which allows distributed applications to communicate and exchange data by sending and receiving messages.

All these models make it possible for one software component to affect the behavior of another component over a network. They are different in that RPC- and ORB-based middleware create systems of tightly-coupled components, whereas MOM-based systems allow for a looser coupling of components. In an RPC- or ORB-based system, when one procedure calls another, it must wait for the called procedure to return before it can do anything else. As mentioned before, in these models, the middleware functions partly as a super-linker, locating the called procedure on a network and using network services to pass function or method parameters to the procedure and then to return results.

MOM-based systems allows communication to happen through the asynchronous exchange of messages, as shown in Figure 1-2.

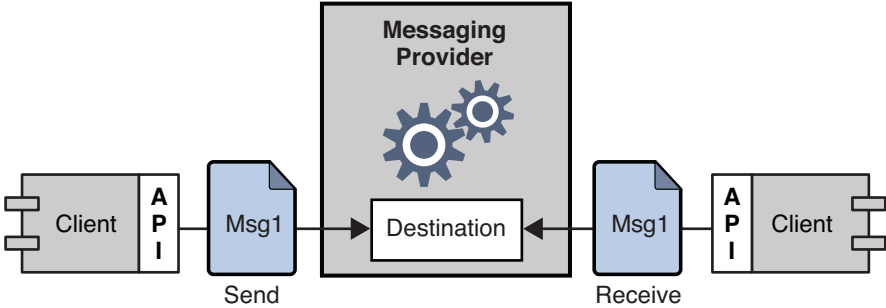


FIGURE 1-2 MOM-Based System

Message Oriented Middleware makes use of messaging provider to mediate messaging operations. The basic elements of a MOM system are clients, messages, and the MOM provider, which includes an API and administrative tools. The MOM provider uses different architectures to route and deliver messages: it can use a centralized message server or it can distribute routing and delivery functions to each client machine. Some MOM products combine these two approaches.

Using a MOM system, a client makes an API call to send a message to a destination managed by the provider. The call invokes provider services to route and deliver the message. Once it has sent the message, the client can continue to do other work, confident that the provider retains the message until a receiving client retrieves it. The message-based model, coupled with the mediation of the provider, makes it possible to create a system of loosely-coupled components. Such a system can continue to function reliably, without downtime, even when individual components or connections fail.

One other advantage of having a messaging provider mediate messaging between clients is that by adding an administrative interface, you can monitor and tune performance. Client applications are thus effectively relieved of every problem except that of sending, receiving, and

processing messages. It is up to the code that implements the MOM system and up to the administrator to resolve issues like interoperability, reliability, security, scalability, and performance.

So far we have described the advantages of connecting distributed components using message-oriented middleware. There are also disadvantages: one of them results from the loose coupling itself. With an RPC system, the calling function does not return until the called function has finished its task. In an asynchronous system, the calling client can continue to load work upon the recipient until the resources need to handle this work are depleted and the called component fails. Of course, these conditions can be minimized or avoided by monitoring performance and adjusting message flow, but this is work that is not needed with an RPC system. The important thing is to understand the advantages and liabilities of each kind of system. Each system is appropriate for different tasks. Sometimes, you will need to combine the two kinds of systems to obtain the exact behavior you need.

Figure 1–3 shows the way a MOM system can enable communication between two RPC-based systems. The left side of the figure shows an application that distributes client, server, and data store components on different networked nodes for improved performance. This is a discount airline reservation system: an end user pays a fee to use this service, which allows it to find the lowest available fare for given destinations and times. The data store holds information about registered users and about airlines that participate in this program. Based on the user's request, logic on the server queries participating airlines for prices, sorts through the information, and presents the three lowest bids to the user. The right side of the picture shows an RPC-based system that represents the ticket/reservation system for any one of the participating airlines. The right side of the picture would be replicated for as many airlines as the discounting service is connected to. For each such airline, the data store would hold information about available flights (seating, flight times, and prices). The server component would update that information in response to data input by the end user. The airline server also subscribes to the MOM service, accepting requests for information from the discount reservation system and returning seating and pricing information. If a customer decides to purchase a discounted ticket on a PanWorld flight, the server component for that system would update the information in the data store and then either generate a ticket for the requester or send a message to the discounting service to generate the ticket.

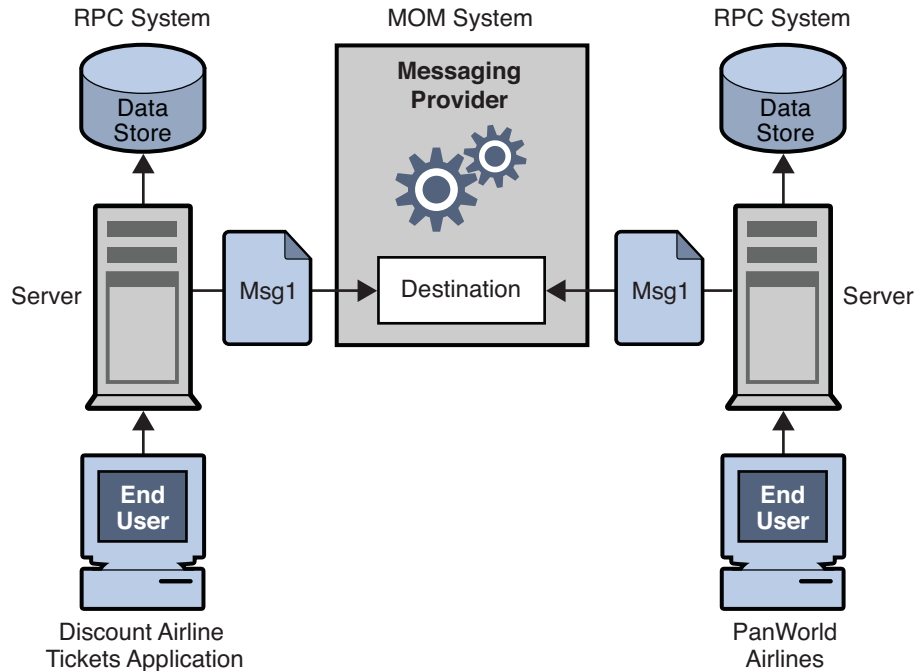


FIGURE 1-3 Combining RPC and MOM Systems

This example illustrates some of the differences between RPC and MOM systems. The difference in the way in which distributed components are coupled has already been mentioned. Another difference is that while RPC systems are often used to distribute and connect client and server components in which the client is often an end-user, with MOM systems, clients are often heterogeneous software components that can only interoperate by means of messaging.

A more serious problem with MOM systems arises from the fact that MOMs are implemented as proprietary products. What happens when your company, which depends on SuperMOM-X acquires a company that uses SuperMOM-Y? To resolve this problem, a standard messaging interface is needed. If both SuperMOM-X and SuperMOM-Y implemented this interface, then applications developed to run on one system could also run on the other. Such an interface should be simple to learn but provide enough features to support sophisticated messaging applications. The Java Message Service (JMS) specification, introduced in 1998, aimed to do just that. The next section describes the basic features of JMS and explains how the standard was developed to embrace common elements of existing proprietary MOM products as well as to allow for differences and further growth.

## JMS as a MOM Standard

The Java Messaging Service specification was originally developed to allow Java applications access to existing MOM systems. Since its introduction, it has been adopted by many existing MOM vendors and it has been implemented as an asynchronous messaging system in its own right.

In creating the JMS specification, its designers wanted to capture the essential elements of existing messaging systems. These included

- The concept of a messaging provider that routes and delivers messages
- Distinct messaging patterns or domains such as point-to-point messaging and publish/subscribe messaging
- Facilities for synchronous and asynchronous message receipt
- Support for reliable message delivery
- Common message formats such as stream, text, and byte

Vendors implement the JMS specification by supplying a *JMS provider* consisting of libraries that implement the JMS interfaces, of functionality for routing and delivering messages, and of administrative tools that manage, monitor, and tune the messaging service. Routing and delivery functions can be performed by a centralized message server or broker, or they could be implemented through functionality that is part of each client's runtime.

Equally, a JMS provider can play a variety of roles: it can be created as a stand-alone product or as an embedded component in a larger distributed runtime system. As a standalone product, it could be used to define the backbone of an enterprise application integration system; embedded in an application server, it could support inter-component messaging. For example, J2EE uses a JMS provider to implement message-driven beans and to allow EJB components to send and receive messages.

To have created a standard that included all features of existing systems would have resulted in a system that was hard to learn and difficult to implement. Instead, JMS defined a least common denominator of messaging concepts and features. This resulted in a standard that is easy to learn and that maximizes the portability of JMS applications across JMS providers. It's important to note that JMS is an API standard, not a protocol standard. It is easy to move a JMS client from one vendor to another. But different JMS vendors typically cannot communicate directly with one another.

The next section describes the basic objects and messaging patterns defined by the JMS specification.

## JMS Messaging Objects and Patterns

In order to send or receive messages, a JMS client must first connect to a JMS provider which is often implemented as a message broker: the connection opens a channel of communication between the client and the broker. Next, the client must set up a session for creating, producing, and consuming messages. You can think of the session as a stream of messages defining a particular conversation between the client and the broker. The client itself is a *message producer* and/or a *message consumer*. The message producer sends a *message* to a *destination* that the broker manages. The message consumer accesses that destination to consume the message. The message includes a header, optional properties, and a body. The body holds the data; the header contains information the broker needs to route and manage the message; and the properties can be defined by client applications or by a provider to serve their own needs in processing messages. Connections, sessions, destinations, messages, producers, and consumers are the basic objects that make up a JMS application.

Using these basic objects, a client application can use two messaging patterns (or domains) to send and receive messages. These are shown in [Figure 1-4](#).

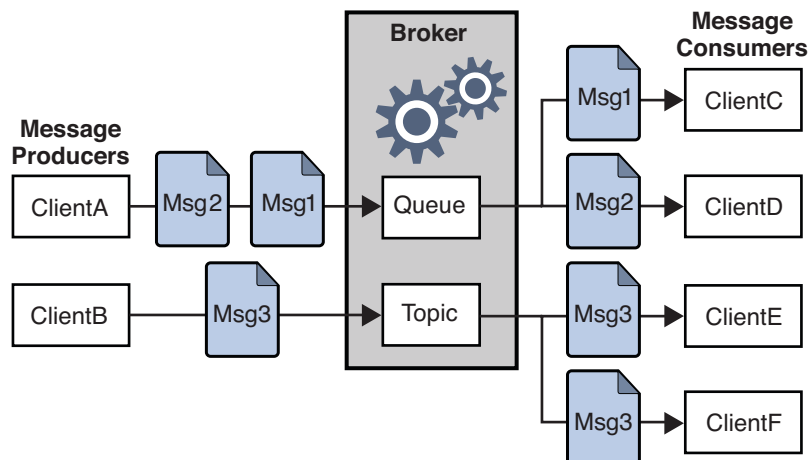


FIGURE 1-4 JMS Messaging Patterns

Clients A and B are message producers, sending messages to clients C, D, and E by way of two different kinds of destinations.

- Messaging between clients A, C, and D illustrates the point-to-point pattern. Using this pattern, a client sends a message to a queue destination from which only one receiver may get it. No other receiver accessing that destination can get that message.
- Messaging between clients B, E, and F illustrates the publish/subscribe pattern. Using this broadcast pattern, a client sends a message to a topic destination from which any number of consuming subscribers can retrieve it. Each subscriber gets its own copy of the message.

Message consumers in either domain can choose to get messages synchronously or asynchronously. Synchronous consumers make an explicit call to retrieve a message; asynchronous consumers specify a callback method that is invoked to pass a pending message. Consumers can also filter out messages by specifying selection criteria for incoming messages.

## Administered Objects

The JMS specification created a standard that combined many elements of existing MOM systems without attempting to exhaust all possibilities. Rather, it sought to set up an extensible scheme that could accommodate differences and future growth. JMS leaves a number of messaging elements up to the individual providers to define and implement. These include load balancing, standard error messages, administrative APIs, security, the underlying wire protocols, and message stores. The next section, “[Message Queue: Elements and Features](#)” on [page 24](#) describes how Message Queue implements many of these elements and how it extends the JMS specification.

Two messaging elements that JMS does not completely define are connection factories and destinations. Although these are fundamental elements in the JMS programming model, there were so many existing and anticipated differences in the ways providers define and manage these objects, that it was neither possible nor desirable to create a common definition. Therefore, these two objects, rather than being created programmatically, are normally created and configured using administration tools. They are then stored in an object store, and accessed by a JMS client through standard JNDI lookups.

- *Connection factory administered objects* are used to generate a client’s connections to the broker. They encapsulate provider-specific information that governs certain aspects of messaging behavior: connection handling, client identification, message header overrides, reliability, and flow control, and so on. Every connection derived from a given connection factory exhibits the behavior configured for that factory.
- *Destination administered objects* are used to reference physical destinations on the broker. They encapsulate provider-specific naming (address-syntax) conventions and they specify the messaging domain within which the destination is used: queue or topic.

JMS clients are not required to look up administered objects; they can create these objects programmatically (which are then stored in the broker’s memory). For quick prototyping, creating these objects programmatically might be easiest. But for deployment in a production environment, looking up administered objects in a central repository makes it much easier to control and manage messaging behavior:

- By using administered objects for connection factory objects, administrators can tune messaging performance by reconfiguring these objects. Performance can be improved without having to recode.
- By using administered objects for physical destinations, administrators can control the proliferation of these destinations on the broker by requiring clients to access these preconfigured objects.

- Administered objects shield developers from provider-specific implementation details and allow the code they develop for one provider to be portable to other providers with little or no change.

The use of administered objects adds a final wrinkle to the picture of the basic JMS application, which is shown in [Figure 1-5](#).

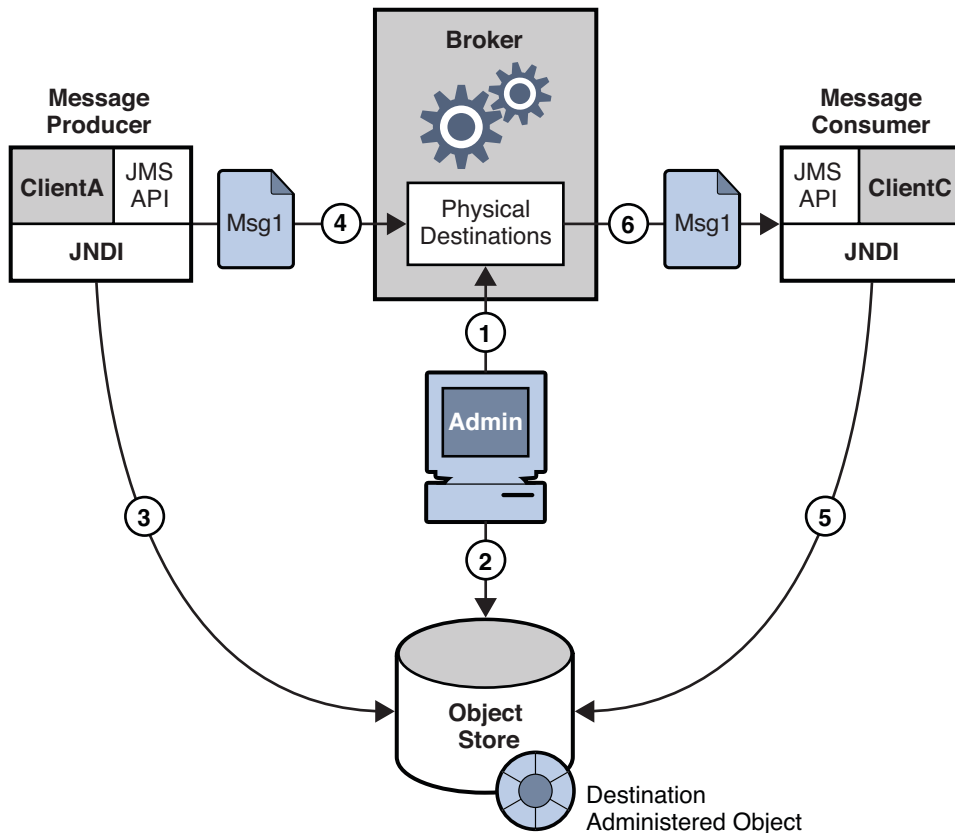


FIGURE 1-5 Basic Elements of a JMS Application

[Figure 1-5](#) shows how a message producer and a message consumer use a destination administered object to access the physical destination to which it corresponds. The marked steps denote the actions that need to be taken by the administrator and by the client applications to send and receive messages using this mechanism.

## ▼ To Use Administered Objects as Destinations

- The administrator creates a physical destination on the broker.

- 2 The administrator creates a destination administered object and configures it by specifying the name of the physical destination to which it corresponds and its type: queue or topic.
- 3 The message producer looks up the destination administered object using a JNDI lookup call.
- 4 The message producer sends a message to the destination.
- 5 The message consumer looks up the destination administered object where it expects to get messages.
- 6 The message consumer gets the message from the destination.

The process of using connection factory administered objects is similar. The administrator creates and configures a connection factory administered object using administration tools. The client looks up the connection factory object and uses it to create a connection.

Although the use of administered objects adds a couple of steps to the messaging process, it also adds robustness and portability to messaging applications.

## Message Queue: Elements and Features

So far we have described the elements of message-oriented middleware and the use of JMS as a way of adding portability to MOM applications. It now remains to describe how Message Queue implements the JMS specification and to introduce the features and tools it uses to provide reliable, secure, and scalable messaging service.

First, like many JMS provider, Message Queue can be used as a stand-alone product or it can be used as an enabling technology, embedded in a J2EE application server to provide asynchronous messaging. [Chapter 5, “Message Queue and J2EE,”](#) describes the role Message Queue plays in J2EE in greater detail. Unlike other JMS providers, Message Queue has been designated as the JMS reference implementation. This designation attests to the fact that Message Queue is a correct and complete JMS implementation. It also guarantees that the Message Queue product will remain current with any future JMS revisions and extensions.

## The Message Queue Service

As a JMS provider, Message Queue offers a *messaging service* that implements the JMS interfaces and that provides administrative services and control. So far, in illustrating JMS providers, the focus has been mainly on the role of the broker in relaying messages. But in fact, a JMS provider must include many elements in addition to the broker to provide reliable, secure, scalable messaging. [Figure 1–6](#) shows the elements that make up the Message Queue message service. These include a variety of connection services (supporting different protocols), administrative tools, and data stores for messaging, monitoring, and user information. The Message Queue service itself includes all elements marked in gray in the figure.



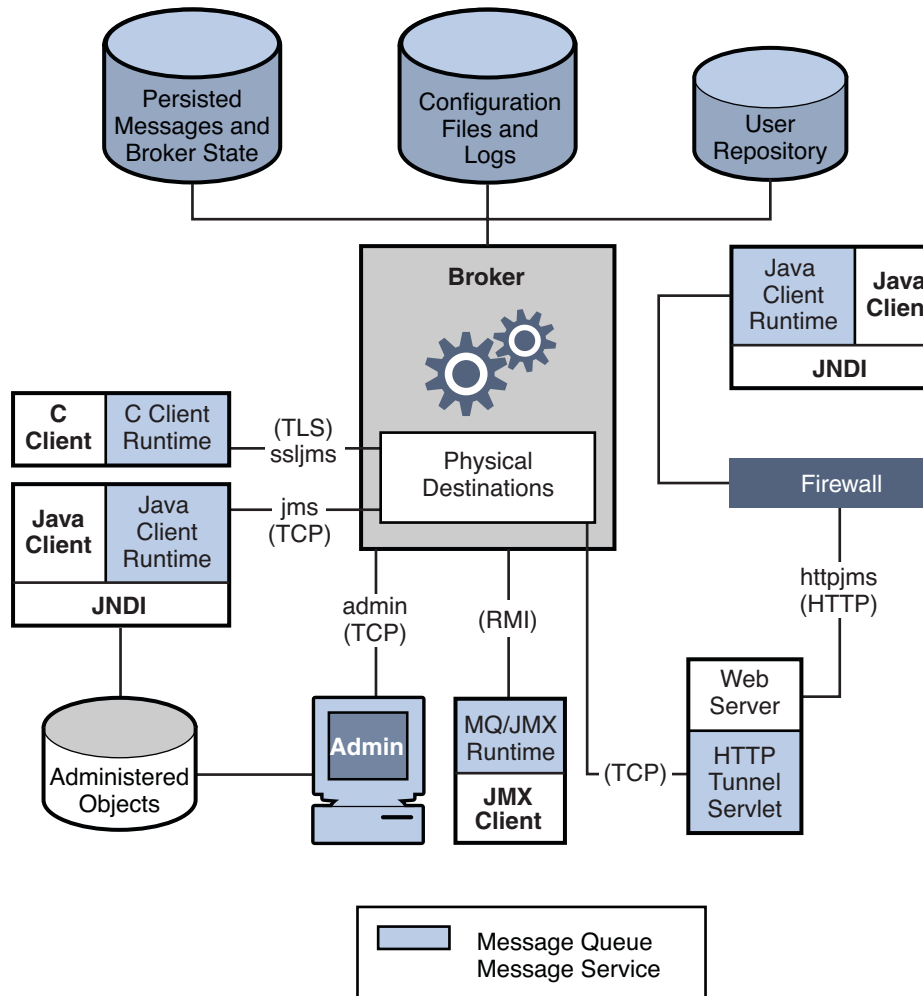


FIGURE 1-6 Message Queue Service

As you can see, a full-featured JMS provider is more complex than the basic JMS model would lead one to suspect. The following sections describe the elements of the Message Queue service shown above. These elements can be divided into three categories: the broker, client runtime support, and administration.

## Connecting to the Broker

As shown in Figure 1-6 both application clients and administration clients can connect to the broker. The JMS specification does not dictate that providers implement any specific wire

protocols. Message Queue services, used by application clients and administration clients to connect to the broker, are currently layered on top of TCP, TLS, HTTP, or HTTPS protocols. (Services layered on top of HTTP allow messages to pass through firewalls.)

- Services that provide JMS support and allow clients to connect to the broker (`jms`, `ssljms`, `http`, or `https`) have a service type of `NORMAL` and are layered on top of TCP, TLS, HTTP, or HTTPS protocols.
- Services that allow administrators to connect to the broker (`admin`, `ssladmin`) have a service type of `ADMIN` and are layered on top of TCP or TLS protocols.

By default, when you start the broker, `jms` and `admin` services are up and running. Additionally, you can configure a broker to run any or all of these connection services. Each service supports specific authentication and authorization (access control) features and each service is multi-threaded, supporting multiple connections.

Should a connection fail, the Message Queue service can automatically retry connecting the client to the same broker or to a different broker if this feature is enabled. For more information, see the description of the automatic reconnect feature in [Appendix B, “Message Queue Features”](#)

Clients can configure connection runtime support when they create the connection factory from which they obtain their connections. Options allow you to specify which brokers to connect to, how to handle reconnection, message flow control, and so on. For additional information about how connections can be configured, see [“Connection Factories and Connections” on page 38](#).

## The Broker

At the heart of the message service is the broker, which routes and delivers messages reliably, authenticates users, and gathers data for monitoring performance.

- To route and deliver messages, the broker places incoming messages in their respective destinations and manages message flow into and out of these destinations.
- To provide reliable delivery, the broker uses a persistent store to save state information and persistent messages until they are received. Should the broker or the connection fail, the saved information allows the broker to restore the broker’s state and to retry operations.
- To provide security for the data being exchanged the broker uses authenticated connections. Optionally data may be encrypted by running over a secure protocol like SSL. The broker also uses and manages a repository that holds information about users and the data or operations they can access. The broker authenticates users requesting services and authorizes the operations they want to carry out by looking up information in this repository.

- To monitor the system, the broker generates metrics and diagnostic information that an administrator can access to measure performance and to tune the broker. Metrics information is also available programmatically to allow applications to adjust message flow and patterns to improve performance.

The Message Queue service provides a variety of administrative tools that the administrator can use to configure broker support. For more information, see “[Administration](#)” on page 28.

## Client Runtime Support

Client runtime support is provided in libraries that you link with when building Message Queue clients. You can think of the client runtime as the bit of the Message Queue service that becomes part of the client. For example, when client code makes an API call to send a message, code in these libraries is invoked that packages the message bits appropriately for the protocol that will be used to relay the message to the physical destination on the broker.

### Java and C Client Support

A JMS provider is only required to support Java clients; however, as [Figure 1–6](#) shows, a Message Queue client can use either the Java or a provider-specific C API to send or receive a message. These interfaces are implemented in Java or C runtime libraries, which do the actual work of creating connections to the broker and packaging the bits appropriately for the connection service requested.

- The Java client runtime supplies Java clients with the objects needed to interact with the broker. These objects include connections, sessions, messages, message producers, and message consumers.
- The C client runtime supplies C clients with the functions and structures needed to interact with the broker. It supports a procedural version of the JMS programming model. C clients cannot use JNDI to access administered objects, but can create connection factories and destinations programmatically.

The Message Queue service provides a C API to enable legacy C and C++ applications to participate in JMS-based messaging. There are a number of differences in the functionality provided by these two APIs; these are documented in “[Java and C Clients](#)” on page 52.

It is important to remember that the JMS specification is a standard for Java clients only. C support is specific to the Message Queue provider and should not be used in client applications that you plan to port to other providers.

### SOAP Support for Java Clients

Message Queue Java clients are also able to send and receive SOAP messages, wrapped as JMS messages. SOAP (Simple Object Access Protocol) allows the exchange of structured data between two peers in a distributed environment. The data exchanged is specified by an XML scheme.

Sun SOAP processing is currently limited to using a point-to-point model and does not guarantee reliability. By wrapping a SOAP message in a JMS message and routing it using the broker, you can take advantage of full featured Message Queue messaging which guarantees reliable delivery and allows you to use the topic as well as the point-to-point domain. Message Queue provides utility routines that a message producer can use to wrap a SOAP message into a JMS message and that a message consumer can use to extract a SOAP message from the JMS message.

“[Working with SOAP Messages](#)” on page 51 gives you a more detailed view of SOAP message processing.

## Administration

The Message Queue service offers command line tools that you can use to do the following:

- Start and configure the broker.
- Create and manage destinations, manage broker connections, and manage broker resources.
- Add, list, update, and deleted administered objects in a JNDI object store.
- Populate and manage a file-based user repository.
- Create and manage a JDBC compliant database for persistent storage.

You can also use a GUI-based administration console to perform the following command-line functions:

- Connect to a broker and manage it.
- Create and manage physical destinations.
- Connect to an object store, add objects to the store, and manage them.

## Scaling the Message Queue Service

As the number of clients or the number of connections grows, you may need to scale the message service to eliminate bottlenecks or to improve performance. The Message Queue message service offers a number of scaling options, depending on your needs. These may be conveniently sorted into the following categories:

- *Vertical scaling* is achieved by adding more processing power and by expanding available resources. You can do this by adding more processors or memory, by switching to a shared thread model, or by running the Java VM in 64 bit mode.

If you are using the point-to-point domain, you can scale the consumer side by allowing multiple consumers to access a queue. Using this approach, you can specify the maximum number of active and backup consumers. The load-balancing mechanism also takes into account a consumer’s current capacity and message processing rate. This is a Message

Queue feature. (The JMS specification defines messaging behavior if only one consumer is accessing a queue; behavior for queues allowing more than one consumer is provider-specific. The Message Queue developer guides provide more information about this scaling option.)

- *Stateless horizontal scaling* is achieved by using additional brokers and redistributing existing clients to these brokers. This approach is easy to implement, but it is appropriate only if your messaging operations can be divided into independent work groups.
- *Stateful horizontal scaling* is achieved by connecting brokers into a *cluster*. In a broker cluster, each broker is connected to every other broker in the cluster as well as to its local application clients. Brokers can be on the same host or distributed across a network. Information about destinations and consumers is replicated on all the brokers in the cluster. Updates to destinations or subscribers is also propagated. Each broker can therefore route messages from producers to which it is directly connected to consumers that are connected to other brokers in the cluster. In situations where backup consumers are used, if one broker or connection fails, messages sent to inaccessible consumers can be forwarded to a backup consumers on another broker.

In the event of broker or connection failure, state information about persistent entities (destinations and durable subscriptions) can get out of sync. For example, if a clustered broker goes down and a destination is created on another broker in the cluster, when the first broker restarts, it will not know about the new destination. Message Queue uses two different models to resolve this problem: conventional clustering and high availability clustering.

- Using conventional clustering, you set broker properties to designate one broker in the cluster to be the *master broker*. This broker is responsible for tracking all changes to destinations and durable subscriptions in a *master configuration file* and for updating brokers in the cluster that are temporarily offline.

When using a master broker, Message Queue only provides service availability, not data availability in the case of broker or connection failure. For example, if a clustered broker becomes unavailable, any persistent messages held by that broker become unavailable until that broker recovers. To get data availability you can use a SunCluster Message Queue agent or you can use high availability clustering, described next. (In the SunCluster case, a persistent store is kept on a shared file system. If a broker fails the Message Queue agent on a second node starts a broker that takes over the shared store. Clients are reconnected to that broker, thereby getting both continuous service and access to persistent data.)

- Using high availability clustering, you set broker properties to specify a highly available database that is shared by all brokers in the cluster. The shared store holds updated information about the state of each broker in the cluster. If one broker fails, another broker assumes ownership of the failed broker's persistent state (in the shared store) and provides uninterrupted service to the failed broker's clients.

For additional information, see [Chapter 4, “Broker Clusters,”](#)

## Message Queue as an Enabling Technology

The Java 2 Platform, Enterprise Edition (J2EE platform) is a specification for a distributed component model in a Java programming environment. One of the requirements of the J2EE platform is that distributed components be able to interact with one another through reliable, asynchronous message exchange. This capacity is furnished by a JMS provider, which can play two roles: it can be used to provide a service and it can support message-driven beans (MDB), a specialized type of Enterprise Java Bean (EJB) component that can consume JMS message.

A J2EE-compliant application server must use a resource adapter furnished by a given JMS provider to use the functionality of that provider. Message Queue provides such a resource adapter. Using the support of a plugged in JMS provider, J2EE components, including MDBs, deployed and running in the application server environment can exchange JMS messages among themselves and with external JMS components. This provides a powerful integration capability for distributed components.

For information on the Message Queue resource adapter, see [Chapter 5, “Message Queue and J2EE,”](#)

## Message Queue Feature Summary

Message Queue has capabilities and features that go far beyond the requirements of the JMS specification. These features enable Message Queue to integrate systems consisting of large numbers of distributed components exchanging many thousands of messages in round-the-clock, mission-critical operations. For a summary of these features, see [Appendix B, “Message Queue Features”](#)

# Client Programming Model

---

This chapter describes the basics of Message Queue client programming. It covers the following topics:

- “Messaging Domains” on page 32
- “Programming Objects” on page 37
- “Producing a Message” on page 42
- “Consuming a Message” on page 43
- “The Request-Reply Pattern” on page 44
- “Reliable Messaging” on page 46
- “A Message’s Journey Through the System” on page 48
- “Java and C Clients” on page 52

This chapter focuses on the design and implementation of Java clients. By and large, C client design roughly parallels Java client design. The final section of this chapter summarizes the differences between Java and C clients. For a detailed discussion of programming Message Queue clients, see *Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients* and *Sun Java System Message Queue 4.1 Developer’s Guide for C Clients*.

Chapter 3, “Message Queue Service,” explains how you can use the Message Queue service to support, manage, and tune messaging performance.

## Design and Performance

The behavior of a Message Queue application depends on many factors: client design, connection configuration, broker configuration, broker tuning, and resource management. Some of these are the responsibility of the developer; others are the concern of the administrator. But in the best of possible worlds the developer should be aware of how the Message Queue service can support and scale the application design, and the administrator should be aware of the design goals when it comes time to tune the application. Messaging behavior can be optimized through redesign as well as through careful monitoring and tuning. Thus, a key aspect of creating a good Message Queue application is for the developer and the

administrator to understand what can be realized at each stage of the application life cycle and to share information about desired and observed behavior.

## Messaging Domains

Messaging middleware allows components and applications to communicate by producing and consuming messages. The JMS API defines two patterns or *messaging domains* that govern this communication: *point-to-point messaging* and *publish/subscribe messaging*. The JMS API is organized to support these patterns. The basic JMS objects: connections, sessions, producers, consumers, destinations, and messages are used to specify messaging behavior in both domains.

### Point-To-Point Messaging

In the point-to-point domain, message producers are called *senders* and consumers are called *receivers*. They exchange messages by means of a destination called a *queue*: senders *produce* messages to a queue; receivers *consume* messages from a queue.

Figure 2–1 shows the simplest messaging operation in the point-to-point domain. MyQueueSender sends Msg1 to the queue destination MyQueue1. Then, MyQueueReceiver obtains the message from MyQueue1.

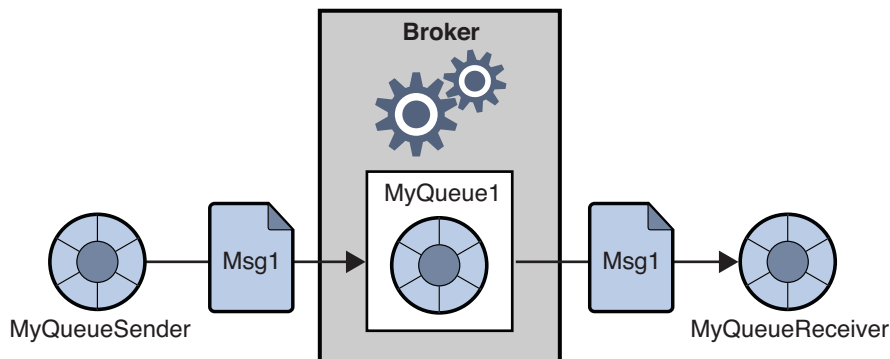


FIGURE 2-1 Simple Point-to-Point Messaging

Figure 2–2 shows a more complex picture of point-to-point messaging to illustrate the possibilities in this domain. Two senders, MyQSender1 and MyQSender2, use the same connection to send messages to MyQueue1. MyQSender3 uses an additional connection to send messages to MyQueue1. On the receiving side, MyQReceiver1 consumes messages from MyQueue1, and MyQReceiver2 and MyQReceiver3, share a connection in order to consume messages from MyQueue1.



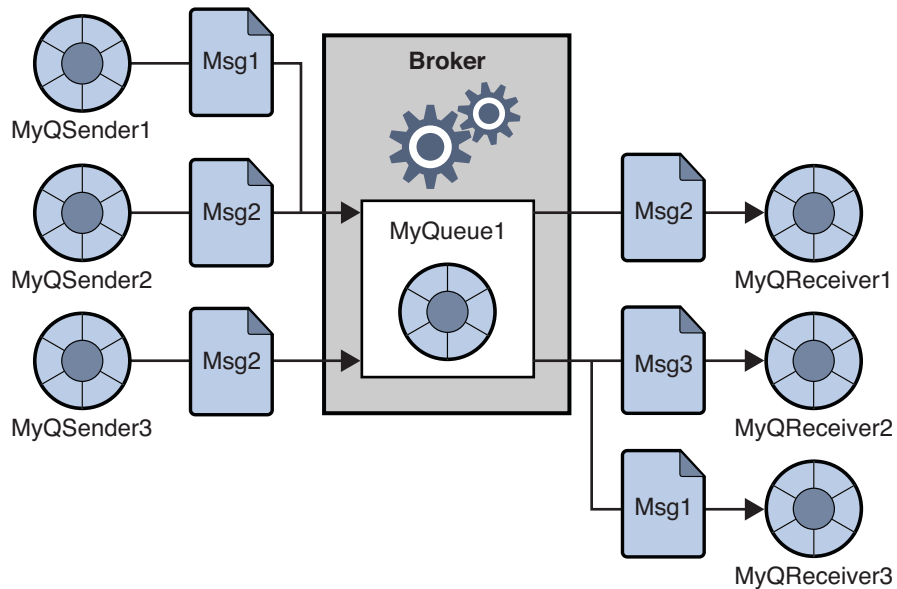


FIGURE 2-2 Complex Point-to-Point Messaging

This more complex picture illustrates a number of additional points about point-to-point messaging.

- More than one producer can send messages to a queue. Producers can share a connection or use different connections, but they can all access the same queue.
- More than one receiver can consume messages from a queue, but each message can only be consumed by one receiver. Thus Msg1, Msg2, and Msg3 are consumed by different receivers. (This is a Message Queue extension.)
- Receivers can share a connection or use different connections, but they can all access the same queue. (This is a Message Queue extension.)
- Senders and receivers have no timing dependencies: the receiver can fetch a message whether or not it was running when the client sent the message.
- Senders and receivers can be added and deleted dynamically at runtime, thus allowing the messaging system to expand or contract as needed.
- Messages are placed on the queue in the order sent, but the order in which they are consumed depends on factors such as message expiration date, message priority, and whether a selector is used in consuming messages.

The point-to-point model offers a number of advantages:

- The fact that multiple receivers can consume messages from the same queue allows you to load-balance message consumption if the order in which messages are received is not important. (This is a Message Queue extension.)

- Messages destined for a queue are always retained, even if there are no receivers.
- Java clients can use a queue browser object to inspect the contents of a queue. They can then consume messages based on the information gained from this inspection. That is, although the consumption model is normally FIFO (first in, first out), consumers can consume messages that are not at the head of the queue if they know what messages they want by using message selectors. Administrative clients can also use the queue browser to monitor the contents of a queue.

## Publish/Subscribe Messaging

In the publish/subscribe domain, message producers are called *publishers* and message consumers are called *subscribers*. They exchange messages by means of a destination called a *topic*: publishers produce messages to a topic; subscribers *subscribe* to a topic and *consume* messages from a topic.

Figure 2–3 shows a simple messaging operation in the publish/subscribe domain. MyTopicPublisher publishes Msg1 to the destination MyTopic. Then, MyTopicSubscriber1 and MyTopicSubscriber2 each receive a copy of Msg1 from MyTopic.

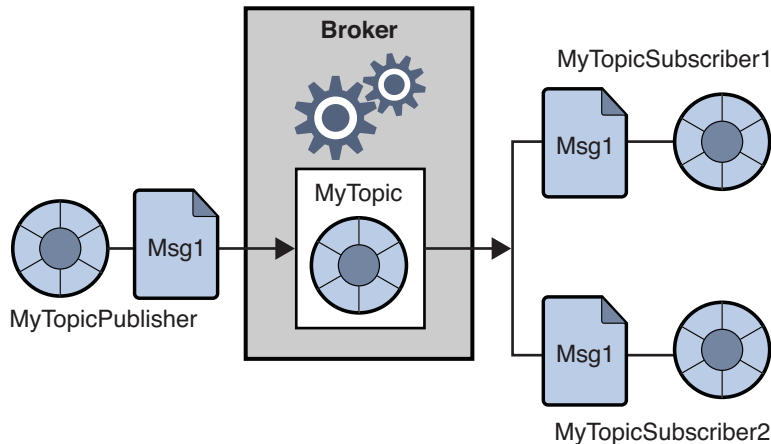


FIGURE 2-3 Simple Publish/Subscribe Messaging

While the publish/subscribe model does not require that there be more than one subscriber, two subscribers are shown in the figure to emphasize the fact that this domain allows you to broadcast messages. All subscribers to a topic get a copy of any message published to that topic.

Subscribers can be non-durable or durable. The broker retains messages for all active subscribers, but it only retains messages for inactive subscribers if these subscribers are durable.

Figure 2–4 shows a more complex picture of publish/subscribe messaging to illustrate the possibilities offered by this pattern. Several producers publish messages to the Topic1 destination. Several subscribers consume messages from the Topic1 destination. Unless, a subscriber is using a selector to filter messages, each subscriber gets all the messages published to the topic of choice. In Figure 2–4, MyTSubscriber2 has filtered out Msg2.

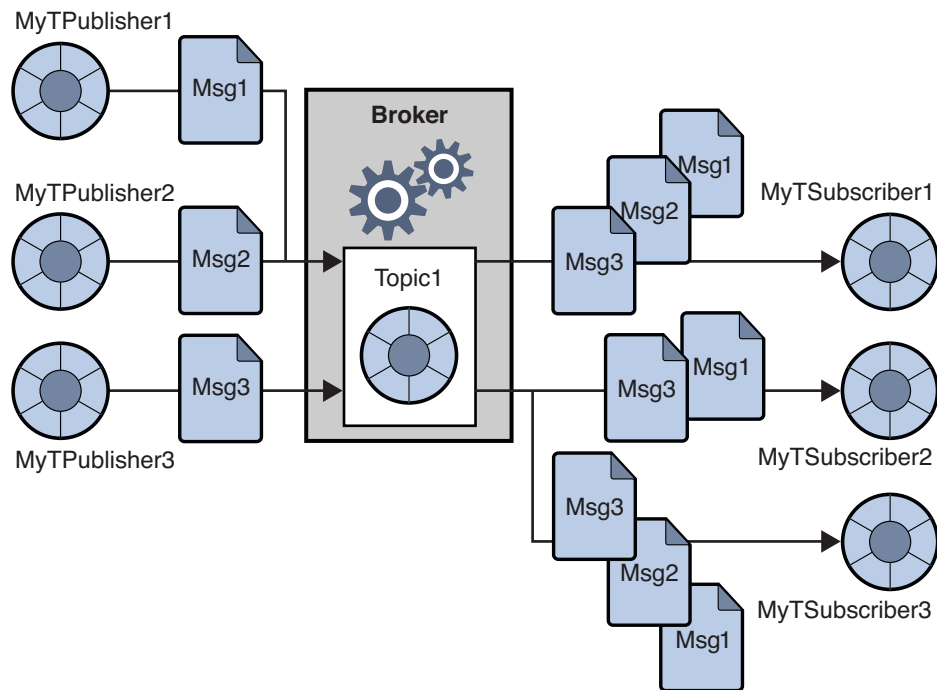


FIGURE 2–4 Complex Publish/Subscribe Messaging

This more complex picture illustrates a number of additional points about publish/subscribe messaging.

- More than one producer can publish messages to a topic. Producers can share a connection or use different connections, but they can all access the same topic.
- More than one subscriber can consume messages from a topic. Subscribers retrieve all messages published to a topic unless they use selectors to filter out messages or the messages expire before they are consumed.
- Subscribers can share a connection or use different connections, but they can all access the same topic.
- Durable subscribers can be active or inactive. The broker retains messages for them while they are inactive.
- Publishers and subscribers can be added and deleted dynamically at runtime, thus allowing the messaging system to expand or contract as needed.

- Messages are published to a topic in the order sent, but the order in which they are consumed depends on factors such as message expiration date, message priority, and whether a selector is used in consuming messages.
- Publishers and subscribers have a timing dependency: a topic subscriber can consume only messages published after it has created the subscription.

The main advantage of the publish/subscribe model is that it allows messages to be broadcast to subscribers.

## Domain-Specific and Unified APIs

The JMS API defines interfaces and classes that you can use to implement either of the point-to-point or the publish/subscribe domains. These are the *domain-specific* APIs shown in columns 2 and 3 of [Table 2–1](#). The JMS API defines an additional *unified* domain, which allows you to program a generic messaging client. The behavior of such a client is determined by the type of the destination to which it produces messages and from which it consumes messages. If the destination is a queue, messaging will behave according to the point-to-point pattern; if the destination is a topic, messaging will behave according to the publish/subscribe pattern.

TABLE 2–1 JMS Programming Domains and Objects

Base Type(Unified Domain)	Point-to-Point Domain	Publish/Subscribe Domain
Destination (Queue or Topic)	Queue	Topic
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

The unified domain was introduced with JMS version 1.1. If you need to conform to the earlier 1.02b specification, you can use the domain-specific API. Using the domain-specific API also provides a clean programming interface that prevents certain types of programming errors: for example, creating a durable subscriber for a queue destination. However, the domain-specific APIs have the disadvantage that you cannot combine point-to-point and publish/subscribe operations in the same transaction or in the same session. If you need to do that, you should choose the unified domain API. See [“The Request-Reply Pattern” on page 44](#) for an example of combining the two domains.

# Programming Objects

The objects used to implement JMS messaging remain essentially the same across programming domains: connection factories, connections, sessions, producers, consumers, messages, and destinations. These objects are shown in [Figure 2-5](#). The figure shows, from the top down, how objects are derived, starting with the connection factory object.

Two of the objects, connection factories and destinations, are shown to reside in an object store. This is to underline the fact that these objects are normally created, configured, and managed as administered objects. We assume that connection factories and destinations are created administratively (rather than programmatically) throughout this chapter.

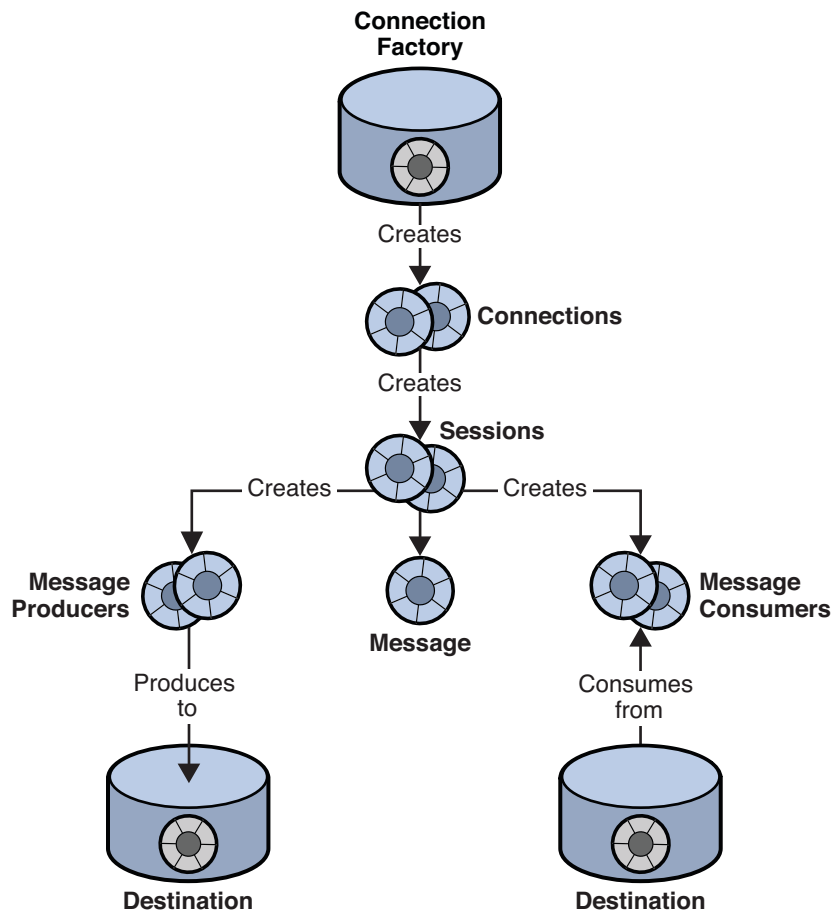


FIGURE 2-5 JMS Programming Objects

Table 2–2 summarizes the steps required to send and receive messages. Note that steps 1 through 6 are the same for senders and receivers.

TABLE 2–2 Producing and Consuming Messages.

Producing a Message	Consuming a Message
1. The administrator creates a connection factory administered object.	
2. The administrator creates a physical destination and the administered object that refers to it.	
3. The client obtains a connection factory object through a JNDI lookup.	
4. The client obtains a destination object through a JNDI lookup.	
5. The client creates a connection and sets any properties that are specific to this connection.	
6. The client creates a session and sets the properties that govern messaging reliability.	
7. The client creates a message producer	The client creates a message consumer
8. The client creates a message.	The client starts the connection.
9. The client sends a message.	The client receives a message.

The following sections describe the objects used by producers and consumers: connections, sessions, messages, and destinations. We will then complete the discussion of JMS objects by describing the production and consumption of messages.

## Connection Factories and Connections

A client uses a *connection factory* object (`ConnectionFactory`) to create a *connection*. A connection object (`Connection`) represents a client's active connection to the broker. It uses the underlying connection service that is either started by default or is explicitly started by the administrator for this client.

Both allocation of communication resources and authentication of the client take place when a connection is created. It is a relatively heavyweight object, and most clients do all their messaging with a single connection. Connections support concurrent use: any number of producers and consumers can share a connection.

When you create a connection factory, you can configure the behavior of all connections derived from it by setting its properties. For Message Queue, these specify the following information:

- The name of the host on which the broker resides, the connection service desired, and the port through which the client wants to access that service.

- How automatic reconnection to the broker should be handled if the connection fails. (This feature reconnects the client to the same (or to a different broker) if a connection is lost. Data failover is not guaranteed: persistent messages and other state information can be lost when reconnecting to a different broker.)
- The ID of a client that needs the broker to track its durable subscription.
- The default name and password of the user attempting the connection. This information is used to authenticate the user and authorize operations if a password is not specified at connection time.
- Whether broker acknowledgements should be suppressed for those clients who are not concerned with reliability.
- How to manage the flow of control and payload messages between the broker and the client runtime.
- How queue browsing should be handled. (Java clients only.)
- Whether certain message header fields should be overridden.

It is possible to override connection factory properties from the command line used to start the client application. It is also possible to override properties for any given connection by setting properties for that connection.

You can use a connection object to create *session* objects, to set up an exception listener, or to obtain JMS version and provider information.

## Sessions

If the connection represents a communication channel between the client and the broker, a session marks a single conversation between them. Mainly, you use a session object to create messages, message producers, and message consumers. When you create a session, you configure reliable delivery through a number of *acknowledgement* options or through transactions. For more information, see [“Reliable Messaging” on page 46](#).

According to the JMS specification, a session is a single-threaded context for producing and consuming messages. You can create multiple message producers and consumers for a single session, but you are restricted to using them serially. The threading implementation varies slightly for Java and C clients. Consult the appropriate developer’s guide for additional information about threading implementation and restrictions.

You can also use a session object to do the following:

- Create and configure destinations for those clients that do not use administered objects to define destinations.
- Create and configure temporary topics and queues; these are used as part of the request-reply pattern. See [“The Request-Reply Pattern” on page 44](#).
- Support transaction processing.

- Define a serial order for producing or consuming messages.
- Serialize the execution of message listeners for asynchronous consumers.
- Create queue browsers. (Java clients only.)

## Messages

A message is composed of three parts: a header, properties, and a body. You must understand this structure in order to compose a message properly and to configure certain messaging behaviors.

### Message Header

A header is required of every JMS message. The header contains ten predefined fields, which are listed and described in [Table 2-3](#).

TABLE 2-3 JMS-Defined Message Header

Header Field	Description
JMSDestination	Specifies the name of the destination object to which the message is sent. (Set by the provider.)
JMSDeliveryMode	Specifies whether the message is persistent. (Set by default by the provider or explicitly by the client for a producer or for an individual message.)
JMSExpiration	Specifies the time when the message will expire. (Set by default by the provider or by the client for a producer or for an individual message.)
JMSPriority	Specifies the priority of the message within a 0 (low) to 9 (high) range. (Set by default by the provider or set explicitly by the client for a producer or for an individual message.)
JMSMessageID	Specifies a unique ID for the message within the context of a provider installation. (Set by the provider.)
JMSTimestamp	Specifies the time when the provider received the message. (Set by the provider.)
JMSCorrelationID	A value that allows a client to define a correspondence between two messages. (Set by the client if needed.)
JMSReplyTo	Specifies a destination where the consumer should send a reply. (Set by the client if needed.)
JMSType	A value that can be evaluated by a message selector. (Set by the client if needed.)
JMSRedelivered	Specifies whether the message has already been delivered but not acknowledged. (Set by the provider.)



As you can see from reading through this table, message header fields serve a variety of purposes: identifying a message, configuring the routing of messages, providing information about message handling, and so on.

One of the most important fields, `JMSDeliveryMode`, determines the reliability of message delivery. This field indicates whether a message is persistent.

- *Persistent messages.* are guaranteed to be delivered and successfully consumed exactly once. Persistent messages are not lost if the message service fails.
- *Non-persistent messages* are guaranteed to be delivered at most once. Non-persistent messages can be lost if the message service fails.

Some message header fields are set by the provider (either the broker or the client runtime) and others are set by the client. Message producers may need to configure header values to obtain certain messaging behaviors; message consumers may need to read header values in order to understand how the message was routed and what further processing it might need.

The header fields (`JMSDeliveryMode`, `JMSEXpiration`, and `JMSPriority`) can be set at three different levels:

- For messages issuing from every connection derived from a connection factory.
- For each message produced.
- For all messages issued by a specific message producer.

If these fields are set at more than one level, values set for the connection factory override those set for the individual message; values set for a given message override those set for the message's producer.

Constant names for message header fields vary with the language implementation. See *Sun Java System Message Queue 4.1 Developer's Guide for Java Clients* or *Sun Java System Message Queue 4.1 Developer's Guide for C Clients* for more information.

## Message Properties

A message can also include optional header fields, called properties, specified as property name and property value pairs. Properties allow clients and providers to extend the message header and can contain any information that the client or the provider finds useful to identify and process a message. Message properties allow a receiving client to ask that only those messages be delivered which fit a given criteria. For instance, a consuming client might indicate an interest for payroll messages concerning part-time employees located in New Jersey. The provider will not deliver messages that do not meet the specified criteria.

The JMS specification defines nine standard properties. Some of these are set by the client and some by the provider. Their names begin with the reserved characters "JMSX." The client or the provider can use these properties to determine who sent a message, the state of the message, how often and when it was delivered. These properties are useful to the provider in routing messages and in providing diagnostic information.

Message Queue also defines message properties, these are used to identify compressed messages and how messages should be handled if they cannot be delivered. For more information see “Managing Message Size” in *Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients*.

## Message Body

The message body contains the data that clients want to exchange.

The type of a JMS message determines what the body may contain and how it should be processed by the consumer, as specified in [Table 2–4](#). The Session object includes a create method for each type of message body.

TABLE 2–4 Message Body Types

Type	Description
StreamMessage	A message whose body contains a stream of Java primitive values. It is filled and read sequentially.
MapMessage	A message whose body contains a set of name-value pairs. The order of entries is not defined.
TextMessage	A message whose body contains a Java string, for example an XML message.
ObjectMessage	A message whose body contains a serialized Java object.
BytesMessage	A message whose body contains a stream of uninterpreted bytes.
Message	A message that contains a header and properties but no body.

Java clients can set a property to have the client runtime compress the body of a message being sent. The Message Queue runtime on the consumer side decompresses the message before delivering it.

## Producing a Message

Messages are sent or published by a message producer, within the context of a connection and session. Producing a message is fairly straightforward: a client uses a message producer object (MessageProducer) to send messages to a physical *destination*, represented in the API by a destination object.

When you create the producer, you can specify a default destination that all the producer’s messages are sent to. You can also specify default values for the message header fields that govern persistence, priority, and time-to-live. These defaults are then used by all messages issuing from that producer unless you override them by specifying an alternate destination when sending the message or by setting alternate values for the header fields for a given message.

The message producer can also implement a request-reply pattern by setting the `JMSReplyTo` message header field. For more information, see [“The Request-Reply Pattern” on page 44](#).

## Consuming a Message

Messages are received by a message consumer, within the context of a connection and session. A client uses a message consumer object (`MessageConsumer`) to receive messages from a specified physical destination, represented in the API as a destination object.

Three factors affect how the broker delivers messages to a consumer:

- Whether consumption is synchronous or asynchronous
- Whether a selector is used to filter incoming messages
- If messages are consumed from a topic destination, whether the subscriber is durable

The other major factor that affects message delivery and client design is the degree of reliability needed for the consumer. See [“Reliable Messaging” on page 46](#).

## Synchronous and Asynchronous Consumers

A message consumer can support either synchronous or asynchronous consumption of messages.

- Synchronous consumption means the consumer explicitly requests that a message be delivered and then consumes it.

Depending on the method used to request messages, a synchronous consumer can choose to wait (indefinitely) until a message arrives, to wait a specified amount of time for a message, or to return immediately if there is no message ready to be consumed. (“Consumed” means the object is immediately available to the client. Messages that were successfully sent but which the broker has not finished processing are not ready to be consumed.)

- Asynchronous consumption means that the message is automatically delivered to a message listener object (`MessageListener`) that has been registered for the consumer. The client consumes the message when a session thread invokes the `onMessage()` method of the message listener object.

## Using Selectors to Filter Messages

A message consumer can use a message selector to have the message service deliver only those messages whose properties match specific selection criteria. You specify this criteria when you create the consumer.

Selectors use an SQL-like syntax to match against message properties. For example,

```
color = "red"  
size > 10
```

Java clients can also specify selectors when browsing a queue; this allows you to see which selected messages are waiting to be consumed.

## Using Durable Subscribers

You can use a session object to create a durable subscriber to a topic. The broker retains messages for these kinds of subscribers even when the subscriber becomes inactive.

Because the broker must maintain state for the subscriber and resume delivery of messages when the subscriber is reactivated, the broker must be able to identify a given subscriber throughout its comings and goings. The subscriber's identity is constructed from the `ClientID` property of the connection that created it and the subscriber name specified when you create the subscriber.

## The Request-Reply Pattern

You can combine producers and consumers in the same connection (or even session when using the unified API). In addition, the JMS API allows you to implement a request-reply pattern for your messaging operations by using temporary destinations.

To set up the request-reply pattern you need to do the following:

1. Create a temporary destination where the consumer can send replies.
2. In the message to be sent, set the `JMSReplyTo` field of the message header to that temporary destination.

When the message consumer processes the message, it examines the `JMSReplyTo` field of the message to determine if a reply is required and sends the reply to the specified destination.

The request-reply mechanism saves the producer the trouble of setting up an administered object for the reply destination and makes it easy for the consumer to respond to the request. This pattern is useful when the producer must be sure that a request has been handled before proceeding.

[Figure 2–6](#) illustrates a request-reply pattern that sends messages to a topic and receives replies in a temporary queue.

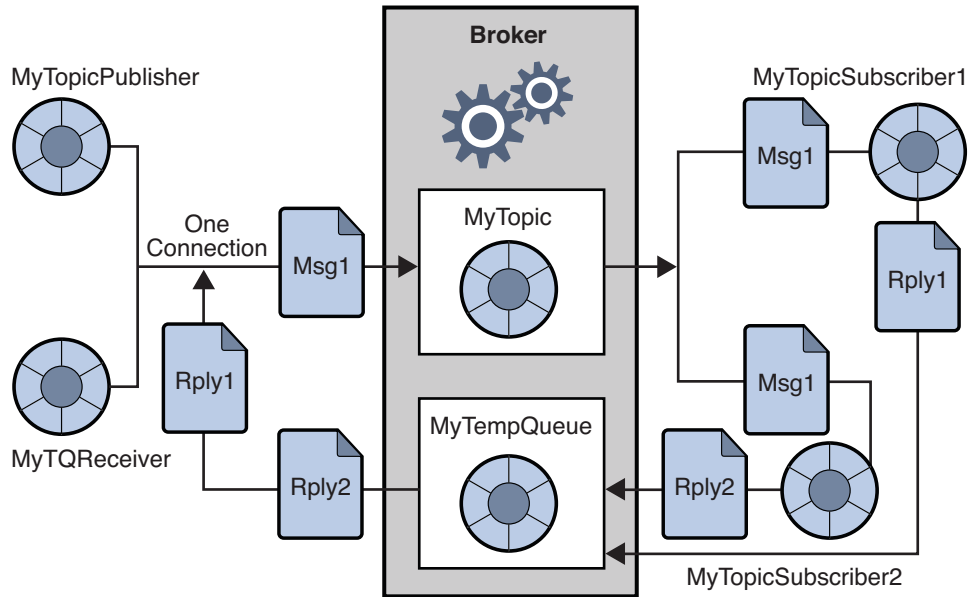


FIGURE 2-6 Request/Reply Pattern

As the figure shows, `MyTopicPublisher` produces `Msg1` to the destination `MyTopic`. `MyTopicSubscriber1` and `MyTopicSubscriber2` receive the message and send a reply to `MyTempQueue`, from where `MyTQReceiver` retrieves it. This pattern might be useful for an application that published pricing information to a large number of clients and which queued their (reply) orders for sequential processing.

Temporary destinations last only as long as the connection that created them. Any producer can send to a temporary destination, but the only consumers that can access temporary destinations are those created by the same connection that created the destination.

Since the request/reply pattern depends on creating temporary destinations, you should not use this pattern in the following cases:

- If you anticipate that the connection creating the temporary destination might terminate before the reply is sent.
- If you need to send a persistent message to a temporary destination.

## Reliable Messaging

Message delivery occurs in two hops: the first hop takes the message from the producer to a physical destination on the broker; the second hop takes the message from that destination to the consumer. Thus, a message can be lost in one of three ways: on its hop to the broker, while it's in broker memory if the broker fails, and on its hop from the broker to the consumer. Reliable delivery guarantees that delivery will not fail at any of these stages. Because non-persistent messages can always be lost if the broker fails, reliable delivery only applies to persistent messages.

Two mechanisms are used to ensure reliable delivery:

- The client can use *acknowledgments* or *transactions* to make sure that message production and consumption is successful.
- The broker can store messages in a persistent store so that if the broker fails before the message is consumed, it can retrieve the stored copy of the message and retry the operation.

The following sections describe these two aspects of ensuring reliability.

## Acknowledgements

*Acknowledgements* are messages sent between the client and the message service to ensure reliable delivery of messages. Acknowledgements are used differently for producers and for consumers.

In the case of message production, the broker acknowledges that it has received the message, placed it in its destination and stored it persistently. The producer's `send()` method blocks until it receives this acknowledgement. These acknowledgements are transparent to the client when persistent messages are sent.

In the case of message consumption, the client acknowledges that it has received delivery of a message from a destination and consumed it, before the broker can delete the message from that destination. JMS specifies different acknowledgement modes that represent different degrees of reliability.

- In the `AUTO_ACKNOWLEDGE` mode, the session automatically acknowledges each message consumed by the client. The session thread blocks, waiting for the broker to confirm that it has processed the client acknowledgement for each consumed message.
- In the `CLIENT_ACKNOWLEDGE` mode, the client explicitly acknowledges after one or more messages have been consumed by calling the `acknowledge()` method of a message object. This causes the session to acknowledge all messages that have been consumed by the session since the previous invocation of the method. The session thread blocks, waiting for the broker to confirm that it has processed the client acknowledgement.

Message Queue extends this mode by providing a method that allows a client to acknowledge receipt of one message only.

- In `DUPS_OK_ACKNOWLEDGE` mode, the session acknowledges after ten messages have been consumed. The session thread does not block waiting for a broker acknowledgement, because no broker acknowledgement is required in this mode. Although this mode guarantees that no message will be lost, it does not guarantee that no duplicate messages will be received, hence its name: `DUPS_OK`.

For clients that are more concerned with performance than reliability, the Message Queue service extends the JMS API by providing a `NO_ACKNOWLEDGE` mode. In this mode, the broker does not track client acknowledgements, so there is no guarantee that a message has been successfully processed by the consuming client. Choosing this mode may give you better performance for non-persistent messages that are sent to non-durable subscribers.

## Transactions

A *transaction* is a way of grouping the production and/or consumption of one or more messages into an atomic unit. The client and broker acknowledgement process described above applies, as well, to transactions. In this case, client runtime and broker acknowledgements operate implicitly on the level of the transaction. When a transaction commits, a broker acknowledgement is sent automatically.

A session can be configured as *transacted*, and the JMS API provides methods for initiating, committing, or rolling back a transaction.

As messages are produced or consumed within a transaction, the message service tracks the various sends and receives, completing these operations only when the JMS client issues a call to commit the transaction. If a particular send or receive operation within the transaction fails, an exception is raised. The client code can handle the exception by ignoring it, retrying the operation, or rolling back the entire transaction. When a transaction is committed, all its operations are completed. When a transaction is rolled back, all successful operations are cancelled.

The scope of a transaction is always a single session. That is, one or more producer or consumer operations performed in the context of a single session can be grouped into a single transaction. Since transactions span only a single session, you cannot have an end-to-end transaction encompassing both the production and consumption of a message.

The JMS specification also supports *distributed* transactions. That is, the production and consumption of messages can be part of a larger, distributed transaction that includes operations involving other resource managers, such as database systems. A transaction manager, like the one supplied by the Java Systems Application Server, must be available to support distributed transactions.

In distributed transactions, a distributed transaction manager tracks and manages operations performed by multiple resource managers (such as a message service and a database manager) using a two-phase commit protocol defined in the Java Transaction API (JTA), XA Resource

API Specification. In the Java world, interaction between resource managers and a distributed transaction manager are described in the JTA specification.

Support for distributed transactions means that messaging clients can participate in distributed transactions through the XAResource interface defined by JTA. This interface defines a number of methods for implementing two-phase commit. While the API calls are made on the client side, the JMS message service tracks the various send and receive operations within the distributed transaction, tracks the transactional state, and completes the messaging operations only in coordination with a distributed transaction manager—provided by a Java Transaction Service (JTS). As with local transactions, the client can handle exceptions by ignoring them, retrying operations, or rolling back an entire distributed transaction.

---

**Note** – Message Queue supports distributed transactions only when it is used as a JMS provider in a Java Enterprise Edition platform. For additional information on how to use distributed transactions, please consult the documentation furnished by your Application Server provider.

---

## Persistent Storage

The other aspect of reliability is ensuring that the broker does not lose persistent messages before they are delivered to consumers. This means that when a message reaches its physical destination, the broker must place it in a persistent *data store*. If the broker goes down for any reason, it can recover the message later and deliver it to the appropriate consumers.

The broker must also persistently store durable subscriptions. Otherwise, in case of failure, it would not be able to deliver messages to durable subscribers who become active after a message has arrived in a topic destination.

Messaging applications that want to guarantee message delivery must specify messages as persistent and deliver them either to topic destinations with durable subscriptions or to queue destinations.

Chapter 3, “Message Queue Service,” describes the default message store supplied by the Message Queue service and how an administrator can set up and configure an alternate store.

## A Message's Journey Through the System

By way of summarizing the material presented so far, this section describes how a message is delivered using the Message Queue service, from a producer to a consumer. In order to paint a complete picture, a further detail is needed: The messages handled by the system in the course of delivery fall into two categories:

- **Payload messages**, which are the messages sent by producers to consumers.



- **Control messages**, which are private messages passed between the broker and the client runtime to ensure that payload messages are successfully delivered and to control the flow of messages across a connection.

Message delivery is illustrated in [Figure 2–7](#).

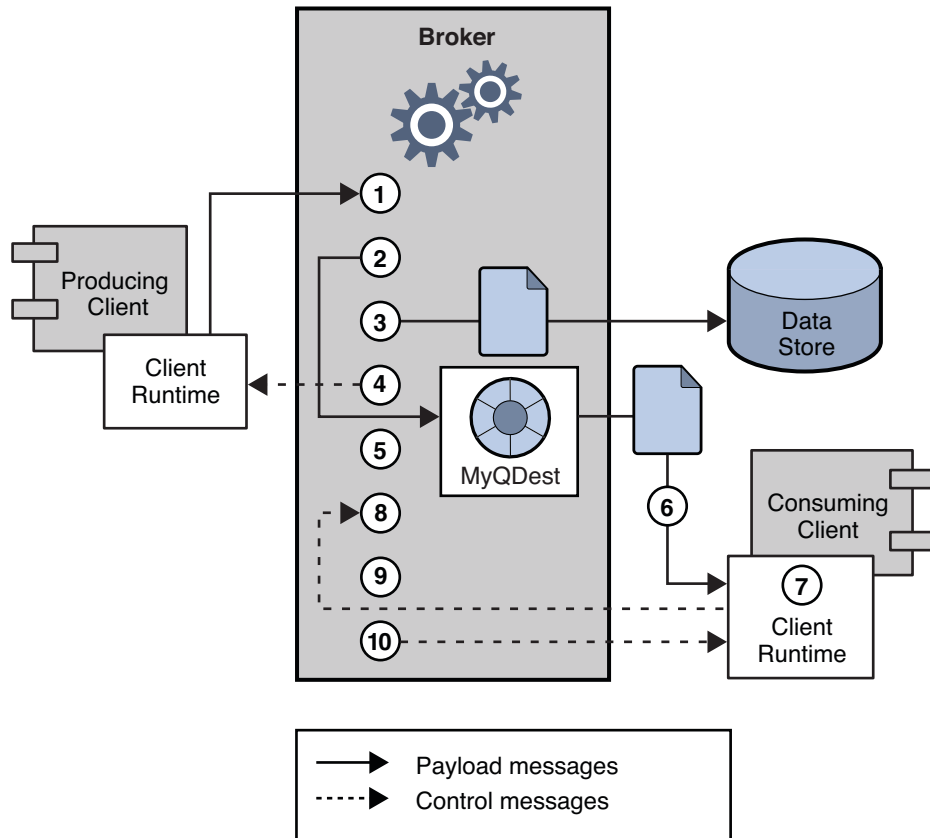


FIGURE 2–7 Message Delivery Steps

Message delivery steps for a persistent, reliably delivered message are as follows:

## Message Production

1. The client runtime delivers the message over the connection from the message producer to the broker.

## Message Handling and Routing

2. The broker reads the message from the connection and places it in the appropriate destination.
3. The broker places the (persistent) message in the data store.
4. The broker acknowledges receipt of the message to the client runtime of the message producer.
5. The broker determines the routing for the message.
6. The broker writes out the message from its destination to the appropriate connection, tagging it with a unique identifier for the consumer.

## Message Consumption

7. The message consumer's client runtime delivers the message from the connection to the message consumer.
8. The message consumer's client runtime acknowledges consumption of the message to the broker.

## Message End-of-Life

9. The broker processes the client acknowledgement, and deletes the (persistent) message when all acknowledgements have been received.
10. The broker confirms to the consumer's client runtime that the client acknowledgement has been processed.

The broker can discard a message before it is consumed if the administrator deletes the message from a destination or if the administrator removes or redefines a durable subscription, thereby causing a message in a topic destination to be removed without it being delivered. In other situations, you might want the broker to store the messages in a special destination called the *dead message queue* rather than discard them. A message is placed on the dead message queue when it expires, when it is removed due to memory limits, or when delivery fails due to the client's throwing an exception. Storing messages in the dead message queue allows you to troubleshoot the system and recover messages in certain situations.

## Working with SOAP Messages

SOAP (see “SOAP Support for Java Clients” on page 27) allows for the exchange of structured data (specified by an XML scheme) between two peers in a distributed environment. Sun’s implementation of SOAP does not currently support reliable SOAP messaging nor does it support publishing SOAP messages. However, you can use the Message Queue service to obtain reliable SOAP messaging and, if desired, to publish SOAP messages. The Message Queue service does not deliver SOAP messages directly, but it allows you to wrap SOAP messages into JMS messages, to produce and consume these messages like normal JMS messages, and to extract the SOAP message from the JMS message.

Message Queue provides SOAP support through two packages: `javax.xml.messaging` and `com.sun.messaging.xml`. You can use classes implemented in these libraries to receive a SOAP message, to wrap a SOAP message into a JMS message, and to extract a SOAP message from a JMS message. The J2EE platform provides the package `java.xml.soap`, which you can use to assemble and disassemble a SOAP message.

### ▼ To Get Reliable SOAP Messaging

- 1 Use the objects defined in the `java.xml.soap` package to construct a SOAP message, or use the servlet defined in the `javax.xml.messaging` package to receive a SOAP message, or use a web service like JAX-RPC to receive a SOAP message.
- 2 Use the Message Transformer utility to convert the SOAP message into a JMS message.
- 3 Send the JMS message to the desired destination.
- 4 Consume the JMS message asynchronously or synchronously.
- 5 After the JMS message is consumed, use the Message Transformer utility to convert it into a SOAP message.
- 6 Use the SAAJ API (defined in the `java.xml.soap` package) to disassemble the SOAP message.

For detailed information about SOAP messages and their processing, see Chapter 5, “Working with SOAP Messages,” in *Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients*.

## Java and C Clients

Message Queue provides a C API to its messaging services to enable legacy C applications and C++ applications to participate in JMS-based messaging.

The JMS programming model is the foundation for the design of a Message Queue C client. *Sun Java System Message Queue 4.1 Developer's Guide for C Clients* explains how this model is implemented by the C data types and functions.

Like the Java interface, the C interface supports the following features:

- Publish/subscribe and point-to-point connections
- Synchronous and asynchronous receives
- CLIENT, AUTO, and DUPS\_OK acknowledgement modes
- Local transactions
- Session recover
- Temporary topics and queues
- Message selectors

However, it is important to understand that the Java Message Service specification is a standard for *Java* clients only; thus the C Message Queue API is specific to the Message Queue provider and cannot be used with other JMS providers. A messaging application that includes a C client cannot be handled by another JMS provider.

The C interface, does not support the following features:

- The use of administered objects
- Map, stream, or object message types
- Consumer-based flow control
- Queue browsers
- JMS application server facilities (Connection Consumer, distributed transactions)
- Receiving or sending SOAP messages
- Receiving or sending compressed JMS messages
- Auto-reconnect or failover, which allows the client runtime to automatically reconnect to a broker if a connection fails
- The NO\_ACKNOWLEDGE mode

# Message Queue Service

---

Message Queue client performance depends on client design and on how you configure and manage the Message Queue service. This chapter provides a more detailed view of the Message Queue service, which was introduced in Chapter 1. It examines its components, it introduces the tools you use to configure these components, and it summarizes the tasks required to manage the message service in different environments. It includes the following sections

- “Component Services” on page 53
- “Administration Tools and Tasks” on page 65
- “Scaling Messaging Operations” on page 70

## Component Services

Figure 3–1 shows the Message Queue service. Chapter 2, “Client Programming Model,” described the programming model and how clients use the Java and C APIs to interact with the client runtime, the part of the message service that is accessible to client applications. This chapter focuses on the components and services of the message service that are accessible to the administrator.

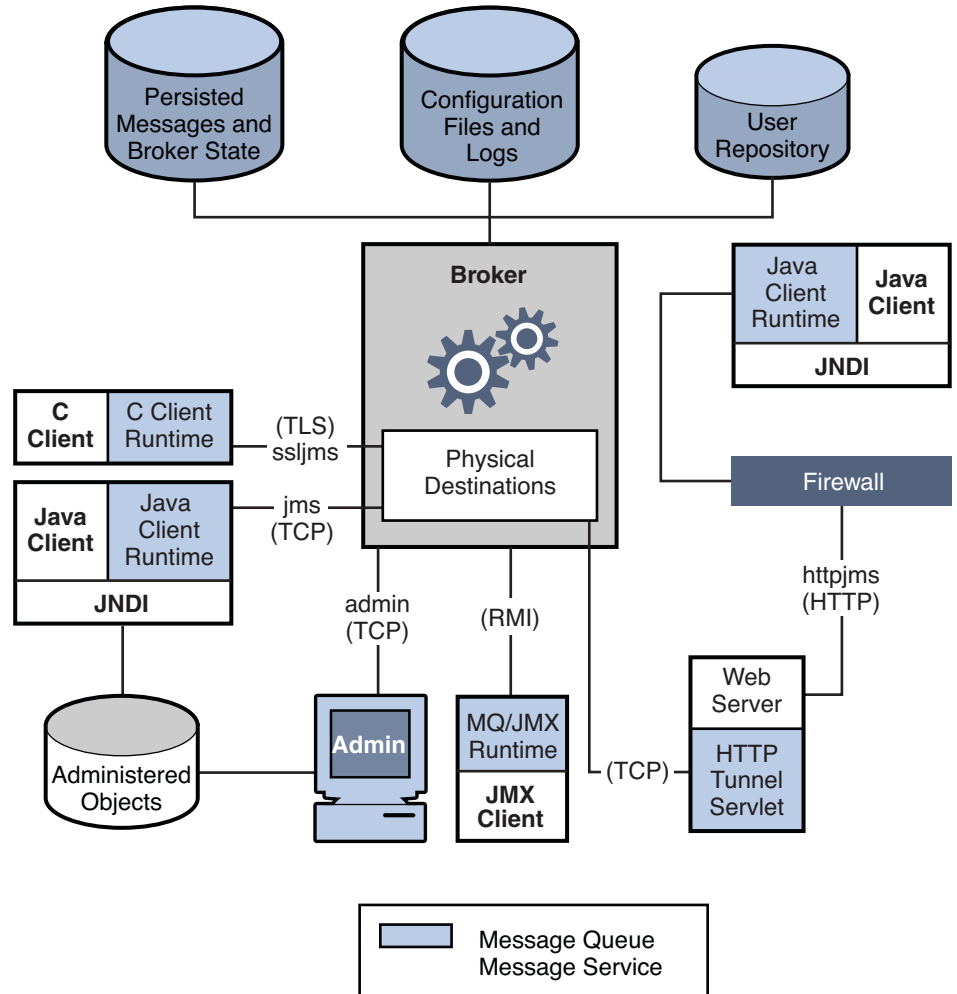


FIGURE 3-1 Message Queue Service

You control the Message Queue service by setting broker properties. These are divided among a number of categories, depending on the services or broker component that is affected by a particular property. Broker services include:

- **Connection services** that manage the physical connections between a broker and its clients that provide transport for incoming and outgoing messages.
- **Routing services** that route and deliver JMS messages as well as control messages used by the message service to support reliable delivery.
- **Persistence services** that manage the writing of data to persistent storage and its retrieval from persistent storage.

- **Security services** that authenticate users connecting to the broker and authorize their actions.
- **Monitoring services** that generate metrics and diagnostic information and write this information to a specified output channel.

The following sections describe each of these services and summarize the properties that you use to customize that service for your particular needs.

Broker properties are defined in different configuration files and can also be defined on the command line used to start the broker. The Chapter 4, “Broker Configuration,” in *Sun Java System Message Queue 4.1 Administration Guide* describes these configuration files and explains the order of precedence by which property values in one file can be used to override values set in a different file. Properties set with the startup command override all other settings.

## Connection Services

You use connection-related properties to configure and manage the physical connections between a broker and its clients. Connection services available for Message Queue clients are introduced in [“Connecting to the Broker” on page 25](#), which describes available connection services: their name, type, and underlying protocol. Connection services are multithreaded and available through dedicated ports that can be dynamically assigned by the broker’s port mapper or statically assigned by the administrator. By default, when you start the broker, the `jms` and `admin` services are up and running.

Because there are two parties to every connection, connection configuration occurs on both sides and needs to be coordinated:

- The client must configure certain attributes of the connection factory object to ask for non-default connection services, hosts, and ports; to specify a list of brokers to connect to in case reconnection to a different broker is required; and to configure reconnection behavior. The client can also specify a ping interval to test for failed connections.
- The administrator, in turn, uses broker properties to activate non-default connection services, to assign static ports if required, to configure threading, and to specify a host to connect to if multiple network cards are used. The administrator can also specify a ping interval to test whether the client is accessible; this is useful in managing resources.

A client can connect to the Message Queue service through a firewall. This can be done either by having the firewall administrator open a specific port and then connecting to that (static) port or by using the HTTP or HTTPS service as summarized in [Appendix B, “Message Queue Features”](#).

Each connection service also supports specific authentication and authorization features. See [“Security Services” on page 60](#) for more information.

## Port Mapper

Connection services are dynamically assigned a port by a common *Port Mapper* that resides at the broker's main port, 7676. When the Message Queue client runtime sets up a connection with the broker, it first contacts the Port Mapper, requesting a port number for the connection service it has chosen.

You can override the Port Mapper by assigning a *static* port number for the `jms`, `ssljms`, `admin` and `ssladmin` connection services when configuring these services. However, static ports are generally used only in special situations, such as in making connections through a firewall and are not generally recommended.

## Thread Pool Management

Each connection service is multithreaded, supporting multiple connections. The threads needed for these connections are maintained by the broker in a pool. How they are allocated depends on the values you specify for the minimum and maximum thread values, and on the threading model you choose.

You can set broker properties to specify a minimum number and maximum number of threads. As threads are needed by connections, they are added to the thread pool for the service supporting that connection. The minimum specifies the number of threads available to be allocated. When the available threads exceeds this minimum threshold, the system will shut down threads as they become free until the minimum is reached again, thereby saving on memory resources. Under heavy loads, the number of threads might increase until the pool's maximum number is reached; at this point, new connections are rejected until a thread becomes available.

The threading model you choose specifies whether threads are dedicated to a single connection or shared by multiple connections:

- In the dedicated model, each connection to the broker requires two threads: one for incoming messages and one for outgoing messages. This limits the number of possible connections but provides high performance.
- In the shared model, connections are processed by a shared thread when sending or receiving messages. Because each connection does not require dedicated threads, this model increases the number of possible connections, but adds some overhead for thread management and thereby impacts performance.

## Destinations and Routing Services

Once clients are connected to the broker, the routing and delivery of messages can proceed. In this phase, the broker is responsible for creating and managing different types of physical destinations, for ensuring a smooth flow of messages, and for using resources efficiently. The broker properties related to routing and destinations are used by the broker to manage these tasks in a way that suits your application's needs.



Remember that a *physical destination* on the broker is a memory location where messages are stored before being delivered to a message consumer. There are four kinds of physical destinations:

- **Admin-created destinations** are created by the administrator using the GUI (`imqadmin`) or the `imqcmd` utility. These correspond either to a logical destination created programmatically or to a destination administered object that is created by the administrator and looked up by the client. You use the `imqcmd` utility to set or update properties for each admin-created destination.
- **Auto-created destinations** are automatically created by the broker whenever a message consumer or producer attempts to access a nonexistent destination. These are typically used during development. You can set a broker property to disallow the creation of such destinations. You set broker properties to configure all auto-create destinations on a particular broker.

An auto-created destination is automatically destroyed by the broker when it is no longer being used: that is, when it has no consumer clients and no longer contains any messages. If a broker restarts, it only recreates this kind of destination if it contains persistent messages.

- Temporary destinations are explicitly created and destroyed programmatically by clients who need a destination where to receive replies to messages. As their name implies, these destinations are temporary and maintained by the broker only for the duration of the connection for which they are created.  
Temporary destinations are not stored persistently and are never recreated when a broker is restarted, but they are visible to administration tools.
- **The *dead message queue*** is a specialized destination, created automatically at broker startup and used to store dead messages for diagnostic purposes. You can set properties for the dead message queue using the `imqcmd` utility.

## Managing Destinations

You use the `imqcmd` utility to manage destinations. Managing a destination involves one or more of the following tasks:

- Creating, pausing, resuming, or destroying a destination
- Listing all destinations on a broker
- Displaying information about the state and properties of a destination
- Displaying metrics information for a destination
- Compacting disk space used to persist messages for a destination
- Updating a physical destination's properties

Management tasks vary with the kind of destination being managed: admin-created, auto-created, temporary, or dead message queue. For example, temporary destinations do not need to be explicitly destroyed; auto created properties are configured using broker configuration properties which apply to all auto-created destinations on that broker.

## Configuring Physical Destinations

For optimal performance, you can set properties when creating or updating physical destinations. Properties that can be set include the following:

- The type and name of the destination.
- Individual and aggregate limits for destinations (the maximum number of messages, the maximum number of total bytes, the maximum number of bytes per message, the maximum number of producers).
- What the broker should do when individual or aggregate limits are exceeded.
- The maximum number of messages to be delivered in a single batch.
- Whether dead messages for a destination should be sent to the dead message queue.
- Whether (in the case of a clustered broker) a destination should be replicated to other brokers in the cluster.

For a queue destination you can also configure the maximum number of back up consumers and you can specify (for clustered brokers) whether delivery to a local queue is preferred.

You can also configure the limits and behavior of the dead message queue. Note, however, that default properties for this queue differ from those of a standard queue.

## Managing Memory

Destinations can consume significant resources, depending on the number and size of messages they handle and on the number and durability of the consumers that register; therefore, they need to be managed closely to guarantee good messaging service performance and reliability.

You can set properties to prevent a broker from being overwhelmed by incoming messages and to prevent the broker from running out of memory. The broker uses three levels of memory protection to keep the message service operating as resources become scarce: destination limits, system-wide limits, and system memory thresholds. Ideally, if destination limits and system-wide limits are set appropriately, critical system-memory thresholds should never be breached.

## Destination Message Limits

You can set destination attributes to manage memory and message flow for each destination. For example, you can specify the maximum number of producers allowed for a destination, the maximum number (or size) of messages allowed in a destination, and the maximum size of any single message.

You can also specify how to broker should respond when any such limits are reached: to slow producers, to throw out the oldest messages, to throw out the lowest-priority messages, or to reject the newest messages.

## System-Wide Message Limits

You can also use properties to set limits that apply to all destinations on a broker: you can specify the total number of messages and the memory consumed by all messages. If any of the system-wide message limits are reached, the broker rejects new messages.

## System Memory Thresholds

Finally, you can use properties to set thresholds at which the broker takes increasingly serious action to prevent memory overload. The action taken depends on the state of memory resources: green (plenty of memory is available), yellow (broker memory is running low), orange (broker is low on memory), red (broker is out of memory). As the broker's memory state progresses from green to red, the broker takes increasingly serious actions:

- It throws out in-memory copies of persistent messages in the data store.
- It throttles back producers of non-persistent messages, eventually stopping the flow of messages into the broker. Persistent message flow is automatically limited by the requirement that each message be acknowledged by the broker.

## Persistence Services

For a broker to recover in case of failure, it needs to recreate the state of its message delivery operations. In order to do this, it must save state information to a data store. When the broker restarts, it uses the saved data to recreate destinations and durable subscriptions, to recover persistent messages, to roll back open transactions, and to recreate its routing table for undelivered messages. It can then resume message delivery.

The Message Queue service supports both file-based and JDBC compliant persistence modules (see [Figure 3-2](#)), and uses file-based persistence by default.

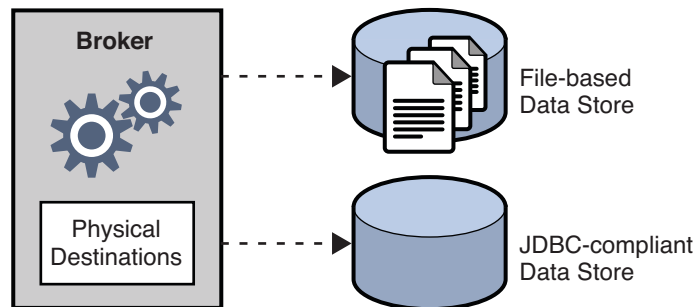


FIGURE 3-2 Persistence Support

## File-Based Persistence

File-based persistence is a mechanism that uses individual files to store persistent data. If you use file-based persistence you can set broker properties to do the following:

- Compact the data store to alleviate fragmentation as messages are added and removed.
- Synchronize the in-memory state with the physical storage device on *every* write. This helps eliminate data loss due to system crashes.
- Manage the allocation of messages to data store files and manage the resources needed for file management and storage.

File-based persistence is generally faster than JDBC-based persistence; however, some users prefer the redundancy and administrative control provided by a JDBC-compliant store.

## JDBC-Based Persistence

JDBC-Based persistence uses a Java Database Connectivity (JDBC™) interface to connect the broker to a JDBC-compliant data store. To have the broker access a data store through a JDBC driver you must do the following:

- Set JDBC-related broker configuration properties. You use these to specify the JDBC driver used, to authenticate the broker as a JDBC user, to create needed tables, and so on.
- Use the `imqdbmgr` utility to create a data store with the proper schema.

Complete procedures for completing these tasks and related configuration properties are detailed in the Chapter 4, “Broker Configuration,” in *Sun Java System Message Queue 4.1 Administration Guide*.

## Security Services

The Message Queue service supports authentication and authorization (access control) for each broker instance, and also supports encryption:

- *Authentication* ensures that only verified users can establish a connection to the broker.
- *Authorization* specifies which users or groups have the right to access resources and to perform specific operations.
- *Encryption* protects messages from being tampered with during delivery over a connection.

Authentication and authorization depend upon a repository that contains information about the users of the messaging system—their names, passwords, and *group* memberships. In addition, to authorize specific operations for a user or group, the broker must check an *access control properties file* that specifies which operations a user or group can perform. You are responsible for setting up the information the broker needs to authenticate users and authorize their actions.

Figure 3–3 shows the components needed by the broker to provide authentication and authorization.

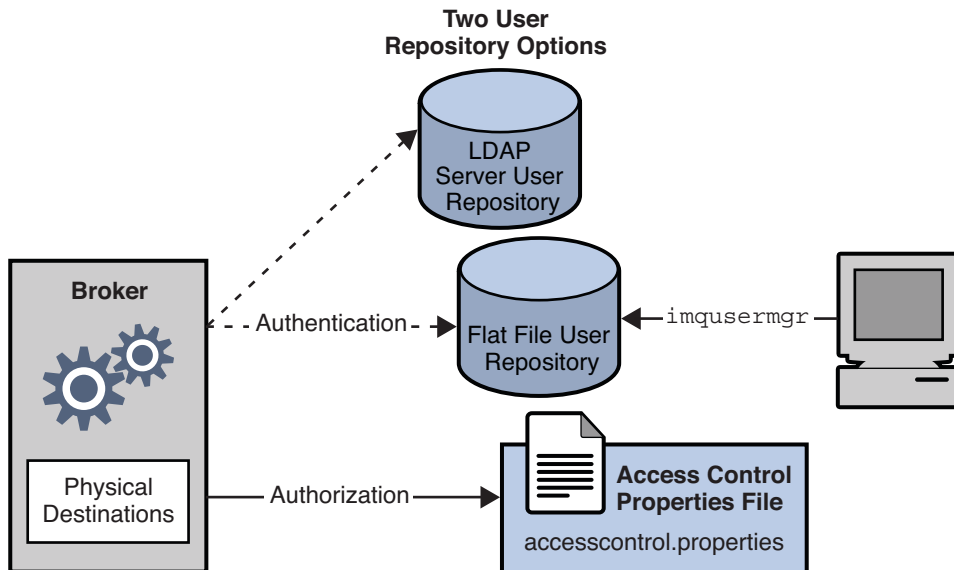


FIGURE 3–3 Security Manager Support

As Figure 3–3 shows, you can store user data in a flat file user repository that is provided with the Message Queue service or you can plug in a pre-existing LDAP repository. You set a broker property to indicate your choice.

- If you choose a flat-file repository, you must use the `imqusermgr` utility to manage the repository. This option is easy to use and built-in.
- If you want to use an existing LDAP server, you use the tools provided by the LDAP vendor to populate and manage the user repository. You must also set properties in the broker instance configuration file to enable the broker to query the LDAP server for information about users and groups.

The LDAP option is better if scalability is important or if you need the repository to be shared by different brokers. This might be the case if you are using broker clusters.

## Authentication and Authorization

When a client requests a connection, the client must supply a user name and password. The broker compares the specified name and password to those stored in the user repository. On transmitting the password from client to broker, the passwords are encoded using either base 64 encoding or message digest (MD5) hashing. MD5 is used for a flat file repository; base 64 is required for LDAP repositories. If using LDAP you may want to use the secure TLS protocol.

You can set broker properties to configure the type of encoding used by each connection service separately or to set the encoding on a broker-wide basis.

When a user attempts to perform an operation, the broker checks the user's name and group membership (from the user repository) against those specified for access to that operation (in the access control properties file). The access control properties file specifies permissions to users or groups for the following operations:

- Connecting to a broker
- Accessing destinations: creating a consumer, a producer, or a queue browser for any given destination or all destinations
- Auto-creating destinations

You set broker properties to specify the following information:

- Whether access control is enabled
- The name of the access control file
- How passwords should be encoded
- How long the system should wait for a client to respond to an authentication request from the broker
- Information required by secure connections

## JAAS-Based Authentication

In addition to the file-based and LDAP-based built-in authentication mechanisms, Message Queue also supports the Java Authentication and Authorization Service (JAAS), which allows you to plug a variety of services into the broker to authenticate Message Queue clients.

JAAS defines an abstraction layer between an application and an authentication mechanism, allowing the desired mechanism to be plugged in with no disruption or change to application code. For the Message Queue service, the abstraction layer lies between the broker and the authentication provider. By setting a few broker properties, it is possible to plug in any JAAS-compliant authentication service and to upgrade or change this service with no disruption or change to broker code.

The service to be plugged in consists of a LoginModule and of logic that performs the authentication. A JAAS configuration file contains the location of the LoginModule. When the broker starts up it locates this file and uses information in the file to determine which LoginModules it will use to perform the authentication. The fact that the broker plugs in an authentication service is transparent to the client; the client continues to pass authentication information to the broker as before and gains access to broker services if the identifying information (user name, password) is authenticated by the plugged in service.

For complete information about JAAS-based authentication, see Chapter 9, "Security," in *Sun Java System Message Queue 4.1 Administration Guide*.

## Encryption

To encrypt messages sent between clients and broker, you need to use a connection service based on the Secure Socket Layer (SSL) standard. SSL provides security at a connection level by establishing an encrypted connection between an SSL-enabled broker and an SSL-enabled client.

You can set broker properties to specify the security properties of the SSL keystore to be used and the name and location of a password file.

## Monitoring Services

The broker includes components for monitoring and diagnosing application and broker performance. These include the following:

- Components that generate data, a metrics generator and broker code that logs events.
- A logger component that writes out information to a number of output channels.
- A message producer that sends JMS messages containing metric information to topic destinations for consumption by JMS monitoring clients.

The general scheme is illustrated in [Figure 3–4](#).

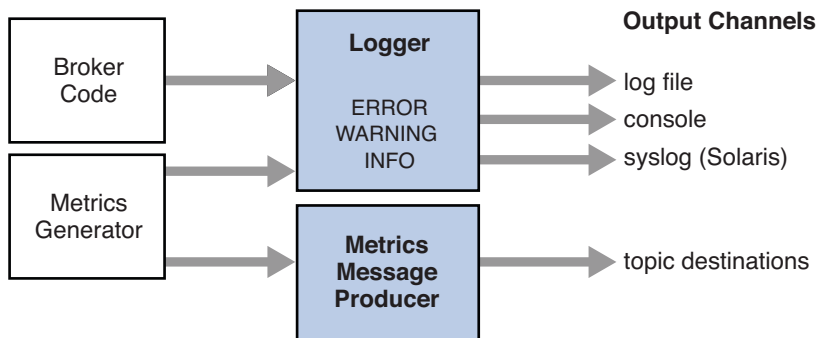


FIGURE 3–4 Monitoring Service Support

## Metrics Generator

The metrics generator provides information about broker activity, such as message flow in and out of the broker, the number of messages in broker memory and the memory they consume, the number of open connections, and the number of threads being used.

You can set broker properties to turn the generation of metric data on and off, and to specify how frequently metrics reports are generated.

## Logger

The Message Queue logger takes information generated by broker code and the metrics generator and writes that information to standard output (the console), to a log file, and, on Solaris™ platforms, to the `syslog` daemon process in case of errors.

You can set broker properties to specify the type of information gathered by the logger as well as the type written to each of the output channels. In the case of a log file, you can also specify the point at which the log file is closed and output is rolled over to a new file. Once the log file reaches a specified size or age, it is saved and a new log file created.

For details about how to configure the logger and how to use it to obtain performance information, see “Configuring and Using Broker Logging” in *Sun Java System Message Queue 4.1 Administration Guide*.

## Metrics Message Producer

The metrics message producer shown in [Figure 3–4](#) receives information from the metrics generator at regular intervals and writes the information into messages, which it then sends to one of a number of metric topic destinations, depending on the type of metric information contained in the message.

Message Queue clients subscribed to these metric topic destinations can consume the messages and process the metric data contained in the messages. This allows developers to create custom monitoring tools to support messaging applications. For details of the metric quantities reported in each type of metrics message, see Chapter 18, “Metrics Reference,” in *Sun Java System Message Queue 4.1 Administration Guide*. For information about how to configure the production of metrics messages, see Chapter 4, “Using the Metrics Monitoring API,” in *Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients* and “Writing an Application to Monitor Brokers” in *Sun Java System Message Queue 4.1 Administration Guide*.

## JES Monitoring Framework Support

Message Queue supports the Sun Java Enterprise System (JES) Monitoring Framework, which allows Java Enterprise System components to be monitored using a common graphical interface. This interface is implemented by a web-based console called the Sun Java System Monitoring Console. If you are running Message Queue along with other JES components, you might find it more convenient to use a single interface to manage all these components.

The JES monitoring framework defines a common data model (CMM) to be used by all JES component products. This model enables a centralized and uniform view of all JES components. Message Queue exposes the following objects to the JES monitoring framework:

- the installed product
- the broker instance name
- the broker port mapper
- each connection service



- each physical destination
- the persistent store
- the user repository

Each one of these objects is mapped to a CMM object whose attributes can be monitored using the JES monitoring console. At runtime, administrators can use the console to view performance statistics, create rules to monitor automatically, and acknowledge alarms. For detailed information about the mapping of Message Queue objects to CMM objects, see the *Sun Java Enterprise System Monitoring Guide*.

To enable JES monitoring, you must do the following

1. Install and configure all the components in your deployment (Message Queue and other components) according to instructions given in the *Sun Java Enterprise System Installation Guide*.
2. Enable and configure the Monitoring Framework for all of your monitored components, as described in the *Sun Java Enterprise System Monitoring Guide*.
3. Install the Monitoring Console on a separate host, start the master agent, and then start the web server, as described in the *Sun Java Enterprise System Monitoring Guide*.

Using the JES Monitoring Framework will not impact broker performance because all the work of gathering metrics is done by the monitoring framework, which pulls data from the broker's existing monitoring data infrastructure.

## Administration Tools and Tasks

This section describes the tools you use to configure Message Queue services and the tasks that you need to complete to support a development or a production environment.

### Administration Tools

[Figure 3–5](#) shows a view of the message service that excludes the client connections and focuses on the broker components and on the tools used to manage these.

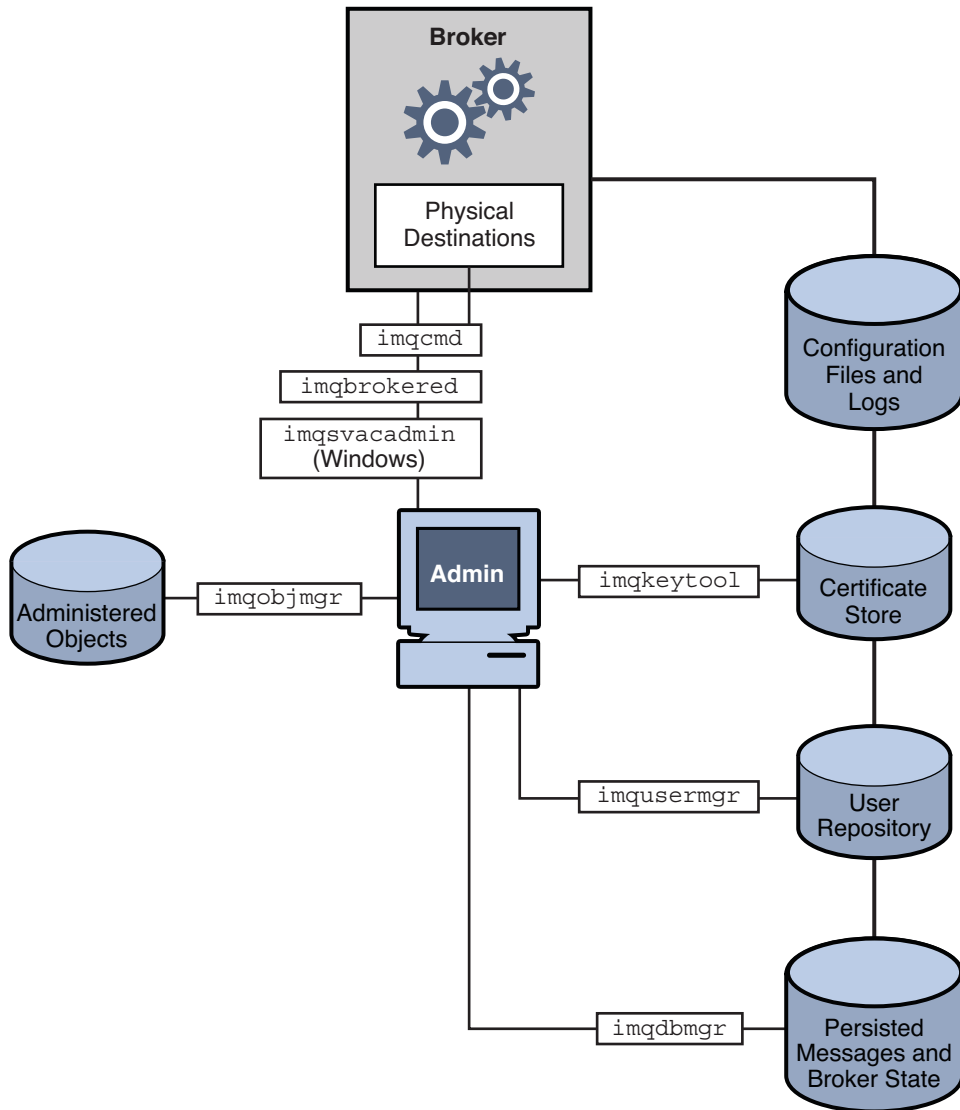


FIGURE 3-5 Administration Tools

You can use the following command-line tools to configure and manage the Message Queue service.

- Use the `imqbrokerd` utility to start the broker. You can use options to the `imqbrokerd` command to specify whether brokers should be connected in a cluster and to specify additional startup configuration information.

- After starting the broker, use the `imqcmd` utility to create, update, and delete physical destinations; to control the broker and its connection services, and to manage the broker's resources.
- Use the `imqobjmgr` utility to add, list, update, and delete administered objects in a JNDI object store.
- Use the `imqusermgr` utility to populate a file-based user repository for user authentication and authorization.
- Use the `imqdbmgr` utility to create and manage a JDBC-compliant database used for persistent storage. (The built-in file store requires no external management.)
- Use the `imqkeytool` utility to generate self-signed certificates used for SSL authentication.
- Use the `imqsvcadm` utility to install, query, and remove the broker as a Windows service.

A GUI-based administration console combines some of the capabilities of the `imqcmd` and `imqobjmgr` utilities. You can use it to do the following:

- Connect to a broker and manage it.
- Create and manage physical destinations.
- Connect to an object store, add objects to the store, and manage them.

## JMX-Based Administration

To serve clients who need a standard programmatic means to monitor and access the broker, Message Queue also supports the Java Management Extensions (JMX) architecture, which allows a client application to manage resources programmatically.

- *Resources* can include applications, services, or devices. In the case of Message Queue, resources include everything that you can manipulate using `imqcmd`: the broker, services, connections, destinations, consumers, producers, and so on.
- *Management* includes the ability to dynamically configure and monitor resources, and the ability to obtain notifications about state changes and error conditions.

JMX-based administration provides dynamic, fine grained, programmatic access to the broker. You can use this kind of administration in a number of ways.

- You can include JMX code in your JMS client application to monitor application performance and, based on the results, to reconfigure the JMS objects you use to improve performance.
- You can write JMX clients that monitor the broker to identify use patterns and performance problems, and you can use the JMX API to reconfigure the broker to optimize performance.
- You can write a JMX client to automate regular maintenance tasks, rolling upgrades, and so on.
- You can write a JMX application that constitutes your own version of `imqcmd`, and you can use it instead of `imqcmd`.

In addition to offering expanded functionality (compared to the JMS API), JMX is also the Java standard for building management applications and is widely used for managing J2EE infrastructure. If your Message Queue client is a part of a larger J2EE deployment, JMX support allows you to use a standard programmatic management framework throughout your J2EE application.

The JMX specification defines an architecture that enables the programmatic management of any distributed resource. This architecture is defined by design patterns, APIs, and various services. Message Queue relies on the implementation of the JMX 1.2 specification, which is part of JDK 1.5.

To manage a Message Queue broker using this architecture, you create an MBean (a managed Java object) that represents the resource to be managed. You manage the underlying resource by configuring the MBean, invoking its operations, or listening for notifications. For complete information about using JMX to manage the Message Queue broker, see *Sun Java System Message Queue 4.1 Developer's Guide for JMX Clients*.

## Supporting a Development Environment

In developing a client component, it's best to keep administrative work to a minimum. The Message Queue product is designed to help you do this and can be used out of the box. It should be enough just to start the broker. The following practices allow you to focus on development:

- Use default implementations of the data store (built-in file persistence), the user repository (file-based), and access control properties file. These are adequate for developmental testing. The default user repository is created with default entries that allow you to use the broker immediately after installation. You can use the default user name (guest) and password (guest) to authenticate a client.
- Use a simple file-system object store by creating a directory for that purpose, and store administered objects there. You can also instantiate administered objects directly in code if you prefer not to create a store at all.
- Use auto-created physical destinations rather than explicitly creating them on the broker. See the appropriate developer's guide for information.

## Supporting a Production Environment

In a production environment, message service management plays a key role in application performance and in meeting the enterprise requirements for scaling, availability, and security. In this environment, the administrator has many more tasks to perform. These can be roughly divided into setup and maintenance operations.

## Setup Operations

Typically, you have to perform the following setup operations:

- Secure administrative access  
Whether you use a file-based or LDAP user repository, make sure that the administrator is in the `admin` group and has a secure password. If necessary, create a secure connection to the broker for the administrator.
- Secure client access  
Whether you use a file-based or LDAP user repository, populate the user repository with the names of users who can access the message service and edit the access control properties file to give them appropriate authorization. If necessary set up SSL-based connection services. To prevent unauthenticated connections, be sure to change the “guest” user’s password.
- Create and configure physical destinations  
Set destination attributes so that the number of messages and the amount of memory allocated for messages can be supported by broker resources.
- Create and configure administered objects.  
If you want to use an LDAP object store, configure and set up the store. Create and configure connection factory and destination administered objects.
- If stateful horizontal scaling is required, create a broker cluster.  
Create a central configuration file and designate a master broker.

## Maintenance Operations

To monitor and control broker resources and to tune application performance, you must do the following after an application has been deployed:

- Support and manage application clients
  - Monitor and manage destinations, durable subscriptions, and transactions
  - Disable auto-create capability
  - Monitor and manage the dead message queue
- Monitor and tune the broker
  - Recover failed brokers
  - Monitor, tune, and reconfigure the broker
  - Manage broker memory resources
  - Expand clusters if necessary
- Manage administered objects  
Create additional administered objects as needed and adjust connection factory attributes to improve performance and throughput.

## Scaling Messaging Operations

The Message Queue service can be scaled horizontally by connecting brokers and allowing them to share state information. This allows any single broker to access remote destinations and to serve a greater number of clients. See [Chapter 4, “Broker Clusters,”](#) for additional information.

# Broker Clusters

---

Message Queue supports the use of *broker clusters*: groups of brokers working together to provide message delivery services to clients. Clusters enable an administrator to scale messaging operations with the volume of message traffic by distributing client connections among multiple brokers.

This chapter discusses the architecture and internal functioning of such broker clusters. It covers the following topics:

- “Cluster Models” on page 71
- “Message Delivery” on page 74
- “Cluster Configuration” on page 80
- “Cluster Synchronization” on page 81
- “Choosing a Clustering Model” on page 82

For complete information about administering broker clusters, see Chapter 8, “Broker Clusters,” in *Sun Java System Message Queue 4.1 Administration Guide*. For information about the effect of reconnection on the client, see “Connection Event Notification” in *Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients* and “Client Connection Failover (Auto-Reconnect)” in *Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients*.

## Cluster Models

Message Queue offers two cluster architectures, depending on the degree of availability desired.

- *Conventional clusters* provide service availability but not data availability. If one broker in a cluster fails, clients connected to that broker can reconnect to another broker in the cluster but may be unable to access some data while they are reconnected to the alternate broker.
- *High availability clusters* provide both service availability and data availability. If one broker in a cluster fails, clients connected to that broker are automatically reconnected to that broker in the cluster which takes over the failed broker's store. Clients continue to operate with all persistent data available to the new broker at all times.

The ability of brokers to work together in a cluster is provided by a *cluster connection service*. This service is configured using broker properties; which properties you configure depend on the model you want to use.

Each cluster model is described next and the sections that follow describe additional concerns and tasks that you need to consider when working with clusters. The chapter ends with a summary of the differences between the two models.

## Conventional Clusters

Figure 4–1 shows Message Queue’s architecture for conventional broker clusters. Each broker within a cluster is directly connected to all the others. Each client (message producer or consumer) has a single *home broker* with which it communicates directly, sending and receiving messages as if that broker were the only one in the cluster. Behind the scenes, the home broker works in concert with the other brokers to provide delivery services for all connected clients.

In a cluster, service availability depends on brokers being able to share information about destinations and durable subscribers. If a clustered broker fails, it is possible that this state information gets out of sync. To guard against this possibility, you can designate one broker within the cluster as the *master broker*. The master broker maintains a *configuration change record* to track changes to the cluster’s persistent entities (destinations and durable subscriptions). This record is used to propagate such change information to brokers that were offline when the changes occurred.



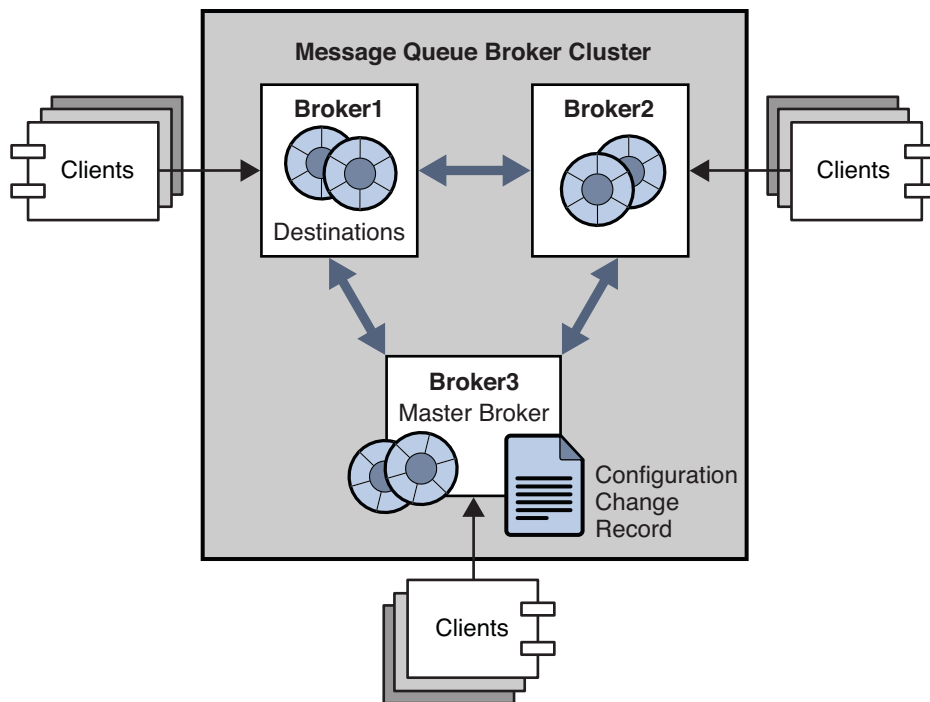


FIGURE 4-1 Cluster Architecture

Following a discussion of the high availability model, this chapter explains how message delivery takes place within a cluster and how the brokers are configured and synchronized.

## High Availability Clusters

High availability clusters provide both service and data availability.

Each broker within a cluster is directly connected to all the others. Each client (message producer or consumer) has a single *home broker* with which it communicates directly, sending and receiving messages as if that broker were the only one in the cluster. Behind the scenes, the home broker works in concert with the other brokers to provide delivery services for all connected clients.

All brokers in a high availability cluster share a common JDBC-based persistent data store that holds dynamic state information (destinations, persistent messages, durable subscriptions, open transactions, and so on) for each broker. If a broker in the cluster fails, another broker takes over the failed broker's lock in the persistent store. Clients connected to the failed broker are reconnected to the broker that has taken over the failed broker's store. The broker that takes over the connection becomes the client's new home broker.

Figure 4–2 shows three brokers connected into a high availability cluster. The dotted line represents the cluster service. In the event that Broker 1 fails or the connection (C1) between clients at Broker 1 is broken, clients are reconnected to Broker 3 using a new connection (C2). Note that all brokers belonging to the high availability cluster are connected to the same highly available database.

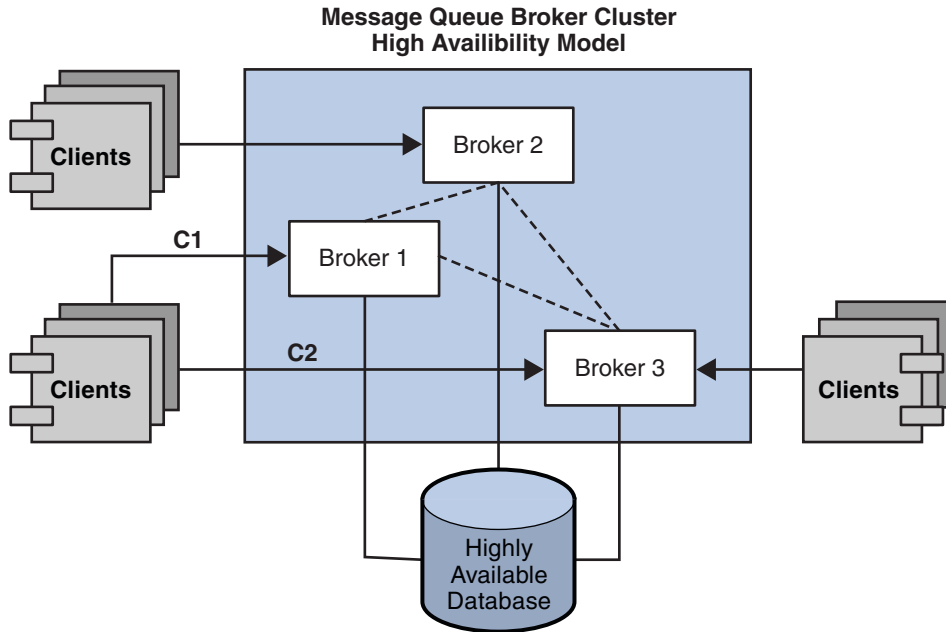


FIGURE 4–2 High Availability Cluster

To configure a high availability cluster you set cluster configuration properties for each broker in the cluster. These specify the cluster id and the broker id in the cluster and they configure the protocol governing the failover process.

## Message Delivery

In a cluster configuration using either model, brokers share information about destinations and message consumers: each broker knows the following information.

- The name, type, and attributes of all physical destinations in the cluster
- The name, location, and interests of each message consumer
- Updates (deletions, additions, or reconfiguration) to the above

This allows each broker to route messages from its own directly connected message producers to remote message consumers. The home broker of a producer has different responsibilities from the home broker of the consumer:

- The producer's home broker is responsible for persisting and routing messages originating from that producer, for logging, for managing transactions, and for processing acknowledgements from consuming clients.
- The consumer's home broker is responsible for persisting information about consumers, for forwarding the message to the consumer, and for letting the producer's broker know whether the consumer is still available and whether the message was successfully consumed.

Clustered brokers work together to minimize message traffic within the cluster; for example, if a remote broker has two identical subscriptions for the same topic destination, the message is sent over the wire only once. You can further reduce traffic by setting a destination property specifying that delivery to local consumers has priority over delivery to remote consumers.

If secure, encrypted message delivery between client and broker is required, you can configure a cluster to provide secure delivery of messages between brokers.

## Destination Attributes

Attributes set for a physical destination on a clustered broker apply to all instances of that destination in the cluster; however, some limits specified by these attributes apply to the cluster as a whole and others to individual destination instances. This behavior is the same for both clustering models. [Table 4-1](#) lists the attributes you can set for a physical destination and specifies their scope.

TABLE 4-1 Properties for Physical Destinations on Clustered Brokers

Property Name	Scope
maxNumMsgs	Per broker. Thus, distributing producers across a cluster, allows you to raise the limit on total unconsumed messages.
maxTotalMsgBytes	Per broker. Thus, distributing producers across a cluster, allows you to raise the limit on total memory reserved for unconsumed messages.
limitBehavior	Global
maxBytesPerMsg	Global
maxNumProducers	Per broker
maxNumActiveConsumers	Global
maxNumBackupConsumers	Global
consumerFlowLimit	Global
localDeliveryPreferred	Global

TABLE 4-1 Properties for Physical Destinations on Clustered Brokers *(Continued)*

Property Name	Scope
isLocalOnly	Global
useDMQ	Per broker

## Clustering and Destinations

How a destination is created (by an administrator, automatically, or as a temporary destination) determines how the destination is propagated in a cluster and how it is handled in the event of connection or broker failure. This behavior is the same for both cluster models. The following subsections examine a few use cases to determine when a destination is created and how it's replicated. These include the following.

- [“Producing to a Queue Using the Reply-To Model” on page 76](#)
- [“Producing to an Auto-Created Destination” on page 77](#)
- [“Publishing to a Topic Destination” on page 78](#)
- [“Handling Destinations in the Event of Connection or Broker Failure” on page 79](#)

### Producing to a Queue Using the Reply-To Model

The figure below shows how destinations are created and replicated when a client produces to a queue and uses the reply-to model.

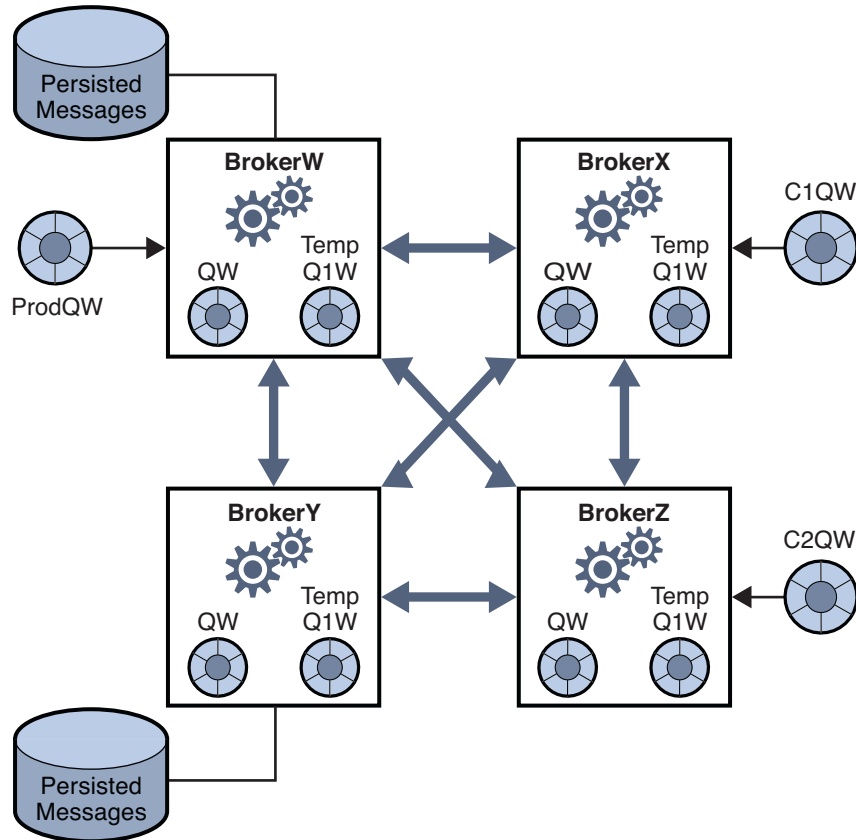


FIGURE 4-3 Replication of Destinations in a Cluster: Queue with Reply-To

1. The administrator creates the physical destination QW. The queue is replicated throughout the cluster at creation time.
2. Producer ProdQW sends a message to queue QW and uses the reply-to model, directing replies to temporary queue TempQ1W. (The temporary queue is created and replicated when an application creates a temporary destination and adds a consumer.)
3. The home broker, BrokerW, persists the message sent to QW and routes the message to the first active consumer that meets the selection criteria for this message. Depending on which consumer is ready to receive the message, the message is delivered either to consumer C1QW (on BrokerX) or to consumer C2QW (on BrokerZ). The consumer receiving the message, sends a reply to the destination TempQ1W.

## Producing to an Auto-Created Destination

The next figure shows how destinations are created and replicated in the case of a producer that sends a message to a destination that does not exist and has to be automatically created.

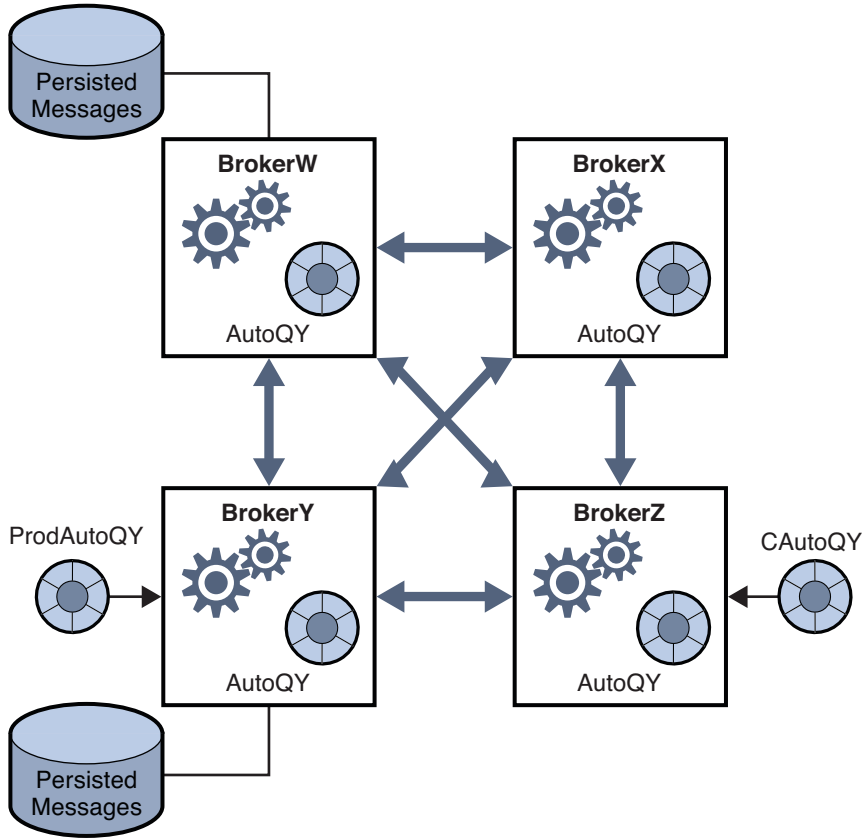


FIGURE 4-4 Replication of Destinations in a Cluster: Auto-Created Destinations

1. Producer ProdAutoQY sends a message to a destination AutoQY that does not exist on the broker.
2. The broker persists the message and creates destination AutoQY.

Auto-created destinations are not automatically replicated across the cluster. Only when a consumer elects to receive messages from a queue AutoQY, would that consumer's home broker create the destination AutoQY and convey the message to the consumer. At the point where one consumer creates the autogenerated destination, the destination is replicated across the cluster. In this example, when the consumer CAutoQY, creates the destination, the replication takes place.

## Publishing to a Topic Destination

The following figure shows how destinations are created and replicated in a cluster when a client publishes a message to a topic destination that is created by the administrator.

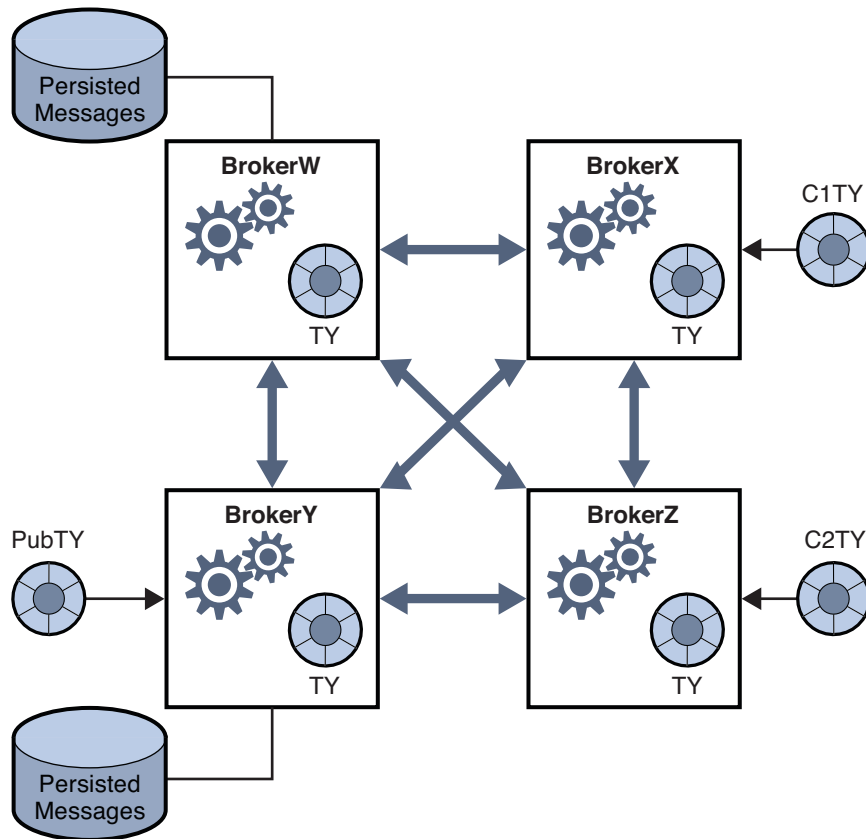


FIGURE 4-5 Replication of Destinations in a Cluster: Publishing to a Topic

1. The administrator creates the physical topic destination TY. The admin-created destination TY is replicated throughout the broker cluster (before the destination is used).
2. Publisher PubTY, sends a message to topicTY.
3. The home broker, BrokerY, persists any messages published to TY and routes the messages to all topic subscribers that match the selection criteria for this message. In this example C1TY and C2TY are subscribed to topicTY.

## Handling Destinations in the Event of Connection or Broker Failure

Table 4-2 explains how different kinds of destinations are replicated and deleted in a cluster.

TABLE 4-2 Handling Destinations in a Cluster

Destination	Propagation, and Deletion
Admin-created	<p>When the destination is created it is propagated in the cluster, and each broker stores information about the destination persistently.</p> <p>The destination is destroyed when the administrator explicitly deletes it.</p> <p>Using the conventional cluster model, if there is a master broker, a record of the creation and deletion is stored in the master broker to allow brokers in the cluster to synchronize state information.</p> <p>Using the high availability cluster model, information synchronized using the shared persistent store.</p>
Temporary	<p>When the destination is created, it is propagated around the cluster.</p> <p>If the consumer associated with the temporary destination is allowed to reconnect, the destination is persistently stored on the consumer's home broker. Otherwise, the destination is never stored. In this case, if the consumer loses its connection, the destination is deleted on all brokers.</p> <p>If the consumer's home broker crashes and the consumer is allowed to reconnect, temporary destinations associated with this consumer are monitored. If the consuming client does not reconnect within a specific period of time, it is assumed that the client has failed and the destination is deleted.</p>
Auto-created	<p>When a producer is created and a destination does not exist, the destination is created on the producer's home broker.</p> <p>When a consumer is created for a destination that does not exist, information about the consumer and the destination is propagated across the cluster.</p> <p>An auto-created destination can be explicitly deleted by an administrator, or it can be automatically deleted</p> <ul style="list-style-type: none"> <li>■ By each broker when there have been no consumers or messages for a given period of time.</li> <li>■ By each broker, when the broker restarts and there are no messages for that destination.</li> </ul>

## Cluster Configuration

Depending on the clustering model used, you must specify appropriate broker properties to enable the Message Queue service to manage the cluster. This information is specified by a set of *cluster configuration properties*. Some of these properties must have the same value for all brokers in the cluster; others must be specified for each broker individually. It is recommended that you place all configuration properties that must be the same for all brokers in one central



*cluster configuration file* that is referenced by each broker at startup time. This ensures that all brokers share the same common cluster configuration information.

See “Configuring Clusters” in *Sun Java System Message Queue 4.1 Administration Guide* for detailed information on cluster configuration properties.

---

**Note** – Although the cluster configuration file was originally intended for configuring clusters, it is also a convenient place to store other (non-cluster-related) properties that are shared by all brokers in a cluster.

---

## Cluster Synchronization

Whenever a cluster’s configuration is changed, information about the change is automatically propagated to all brokers in the cluster. A cluster configuration changes when one of the following events occurs:

- A destination on one of the cluster’s brokers is created or destroyed.
- The properties of a destination are changed.
- A message consumer is registered with its home broker.
- A message consumer is disconnected from its home broker (whether explicitly or through failure of the client, the broker, or the network).
- A message consumer establishes a durable subscription to a topic.

Information about these changes is propagated immediately to all brokers in the cluster that are online at the time of the change. However, a broker that is offline (one that has crashed, for example) will not receive notice of the change when it occurs. How such a broker is resynchronized with the cluster depends on the clustering model used.

Using high availability clustering, synchronization is enabled by the shared persistent store. When a broker that has been offline rejoins the cluster (or when a new broker is added to the cluster) it is able to access the most current information simply by accessing the shared persistent database.

Using conventional clustering, to accommodate offline brokers, the Message Queue service maintains a *configuration change record* for the cluster, recording all persistent entities (destinations and durable subscriptions) that have been created or destroyed. When an offline broker comes back online (or when a new broker is added to the cluster), it consults this record for information about destinations and durable subscribers, then exchanges information with other brokers about currently active message consumers.

One broker in the cluster, designated as the *master broker*, is responsible for maintaining the configuration change record. Because other brokers cannot complete their initialization without the master broker, it should always be the first broker started within the cluster. If the

master broker goes offline, configuration information cannot be propagated throughout the cluster, because other brokers cannot access the configuration change record. Under these conditions, you will get an exception if you try to create, reconfigure, or destroy a destination or a durable subscription or attempt a related operation such as reactivating a durable subscription. (Non-administrative message delivery continues to work normally, however.) The use of a master broker and a configuration change record is optional. They are only required if you are concerned with cluster synchronization after cluster configuration changes or a broker failure.

## Choosing a Clustering Model

The following table summarizes the differences between the two models. Use this information in deciding which model to use or in switching from one model to another.

TABLE 4-3 Clustering Model Differences

Functionality	Conventional	High Availability
Performance	Slightly faster than high availability model.	Slightly slower than conventional model
Service availability	Yes, but some operations are not possible when master broker is down.	Yes.
Data availability	No, when a broker in the cluster id down.	Yes at all times.
Transparent failover recovery	May not be possible if failover occurs during a commit. Rare.	May not be possible if failover occurs during a commit and the client cannot reconnect to any other broker in the cluster. Extremely rare.
Configuration	Done by setting appropriate cluster configuration broker properties.	Done by setting appropriate cluster configuration broker properties.
Additional requirements	None.	Highly available database.

# Message Queue and J2EE

---

The Java 2 Platform, Enterprise Edition (J2EE platform) is a specification for a standard server platform hosting multi-tier and thin client enterprise applications. One of the requirements of the J2EE platform is that distributed components be able to interact through reliable, asynchronous messaging. This interaction is enabled through the use of a JMS provider. In fact, Message Queue is the reference JMS implementation for the J2EE platform.

This chapter explores the ramifications of implementing JMS support in a J2EE platform environment. The chapter covers the following topics:

- “JMS/J2EE Programming: Message-Driven Beans” on page 83
- “J2EE Application Server Support” on page 85

For additional information about using Message Queue as a JMS provider for J2EE compliant application servers, see Chapter 17, “JMS Resource Adapter Property Reference,” in *Sun Java System Message Queue 4.1 Administration Guide*.

## JMS/J2EE Programming: Message-Driven Beans

In addition to the general JMS client programming model introduced in [Chapter 2, “Client Programming Model,”](#) there is a more specialized adaptation of a JMS client used in the context of J2EE platform applications. This specialized client is called a *message-driven bean* and is one of a family of Enterprise JavaBeans (EJB) components described in the EJB 2.0 (and later) Specification (<http://java.sun.com/products/ejb/docs.html>).

Message-driven beans provide asynchronous messaging; other EJB components (session beans and entity beans) can only be called synchronously, through standard EJB interfaces. However, enterprise applications often need asynchronous messaging, to allow server-side components to communicate without tying up server resources. Any application whose server-side components must respond to application events needs an EJB component that can receive and consume messages without being tightly coupled to the message producer. In enterprise applications, this capability must also scale under increasing load.

A message-driven bean (MDB) is an EJB component supported by a specialized EJB container, that provides distributed services for the components it supports.

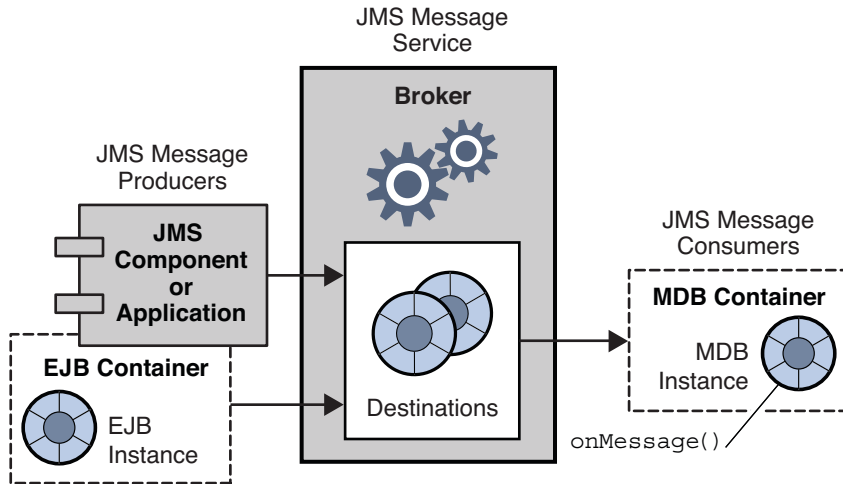


FIGURE 5-1 Messaging with MDBs

- A JMS message driven bean is an EJB that implements the `JMSMessageListener` interface. The `onMessage` method (written by the MDB developer) is invoked when the MDB container receives a message. The `onMessage()` method consumes the message, just as the `onMessage()` method of a standard `MessageListener` object would. You do not remotely invoke methods on MDBs—as you do on other EJB components: therefore there are no home or remote interfaces associated with them. The MDB can consume messages from a single destination. The messages can be produced by standalone JMS applications, JMS components, EJB components, or Web components, as shown in [Figure 5-1](#).
- A specialized EJB container supports the MDB. It creates instances of the MDB and sets them up for asynchronous consumption of messages. The container sets up a connection with the message service (including authentication), creates a pool of sessions associated with a given destination, and manages the distribution of messages among the pooled sessions. Since the container controls the life cycle of MDB instances, it manages the pool of MDB instances to accommodate incoming message loads.

Associated with an MDB is a deployment descriptor that specifies the attributes for the connection factory and destinations used by the container in setting up message consumption. The deployment descriptor can also include other information needed by deployment tools to configure the container. Each such container supports instances of a single MDB.

## J2EE Application Server Support

In J2EE architecture, EJB containers are hosted by J2EE application servers. An application server provides resources needed by the various containers: transaction managers, persistence managers, name services, and, in the case of messaging and MDBs, a JMS provider.

In the Sun Java System Application Server, JMS messaging resources are provided by Sun Java System Message Queue:

- For Sun Java System Application Server 7.0, a Message Queue messaging system is integrated into the application server as its native JMS provider.
- For the Sun J2EE 1.4 Application Server, Message Queue is plugged into the application server as an embedded JMS resource adapter.

For future releases of the Application Server, Message Queue will be plugged into the application server using standard resource adapter deployment and configuration methods.

For information about J2EE architecture, see the J2EE Platform Specification located at <http://java.sun.com/javaee/downloads/index.jsp>.

## JMS Resource Adapter

A *resource adapter* is a standardized way of plugging additional functionality into an application server that complies with J2EE 1.4. The standard, defined by the J2EE Connector Architecture (J2EECA) 1.5 specification, allows an application server to interact with external systems in a standard way. External systems can include enterprise information systems (EIS), as well as messaging systems: for example, a JMS provider. Message Queue includes a JMS resource adapter that allows application servers to use Message Queue as a JMS provider.

Plugging a JMS resource adapter into an application server allows J2EE components deployed and running in the application server to exchange JMS messages. The JMS connection factory and destination administered objects needed by these components can be created and configured using J2EE application server administration tools.

Other administrative operations, however, such as managing a broker and physical destinations, are not included in the J2EECA specification and can be performed only through provider specific tools.

The Message Queue resource adapter is integrated in the Sun J2EE 1.4 application server. However, it has not yet been certified with any other J2EE 1.4 application servers.

The Message Queue resource adapter is a single file (`imqjmsra.rar`) located in a directory that depends on the operating system (see Chapter 17, “JMS Resource Adapter Property Reference,” in *Sun Java System Message Queue 4.1 Administration Guide*). The `imqjmsra.rar` file contains the resource adapter deployment descriptor (`ra.xml`) as well as the JAR files needed by the application server in order to use the adapter.

You can use the Message Queue resource adapter in any J2EE-1.4-compliant application server by following the resource adapter deployment and configuration instructions that come with that application server. As commercial J2EE 1.4 application servers become available and the Message Queue resource adapter is certified for those application servers, Message Queue documentation will provide specific information on the relevant deployment and configuration procedures.

# Message Queue Implementation of Optional JMS Functionality

---

The JMS specification indicates certain items that are optional: each JMS provider (vendor) chooses whether to implement them. This appendix describes how the Message Queue product handles JMS optional items.

[Table A-1](#) describes how the Message Queue service handles JMS optional items.

## Optional Features

TABLE A-1 Optional JMS Functionality

Section in JMS Specification	Description and Message Queue Implementation
3.4.3 JMSMessageID	<p>“Since message IDs take some effort to create and increase a message’s size, some JMS providers may be able to optimize message overhead if they are given a hint that message ID is not used by an application. JMS Message Producer provides a hint to disable message ID.”</p> <p><b>Message Queue implementation:</b> Product does not disable Message ID generation (any <code>setDisableMessageID()</code> call in <code>MessageProducer</code> is ignored). All messages will contain a valid <code>MessageID</code> value.</p>
3.4.12 Overriding Message Header Fields	<p>“JMS does not define specifically how an administrator overrides these header field values. A JMS provider is not required to support this administrative option.”</p> <p><b>Message Queue implementation:</b> The Message Queue product supports administrative override of the values in message header fields through configuration of the client runtime (see <a href="#">“Message Header” on page 40</a>).</p>

TABLE A-1 Optional JMS Functionality (Continued)

Section in JMS Specification	Description and Message Queue Implementation
3.5.9 JMS Defined Properties	<p>“JMS Reserves the ’JMSX’ Property name prefix for JMS defined properties.”“Unless noted otherwise, support for these properties is optional.”</p> <p><b>Message Queue implementation:</b> The JMSX properties defined by the JMS 1.1 specification are supported in the Message Queue product (see Appendix B, “Stability of Message Queue Interfaces,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>).</p>
3.5.10 Provider-specific Properties	<p>“JMS reserves the ’JMS_&lt;vendor_name &gt;’ property name prefix for provider-specific properties.”</p> <p><b>Message Queue implementation:</b> The purpose of the provider-specific properties is to provide special features needed to support JMS use with provider-native clients. They should not be used for JMS to JMS messaging.</p>
4.4.8 Distributed Transactions	<p>“JMS does not require that a provider support distributed transactions.”</p> <p><b>Message Queue implementation:</b> Distributed transactions are supported in this release of the Message Queue product (see “<a href="#">Transactions</a>” on page 47).</p>
4.4.9 Multiple Sessions	<p>“For PTP &lt;point-to-point distribution model&gt;, JMS does not specify the semantics of concurrent <code>QueueReceivers</code> for the same queue; however, JMS does not prohibit a provider from supporting this.” See section 5.8 of the JMS specification for more information.</p> <p><b>Message Queue implementation:</b> The Message Queue implementation supports queue delivery to multiple consumers. For more information, see “<a href="#">Point-To-Point Messaging</a>” on page 32.</p>



# Message Queue Features

---

The Message Queue service fully implements the JMS 1.1 specification for reliable, asynchronous, flexible message delivery. For information about JMS compliance-related issues, see [Appendix A, “Message Queue Implementation of Optional JMS Functionality”](#). However, Message Queue has capabilities and features that exceed JMS requirements. You can use these features to integrate and monitor systems consisting of large numbers of distributed components exchanging many thousands of messages in round-the-clock, mission-critical operations.

This book has introduced these features in the process of describing the Message Queue service. For your convenience, this appendix provides a summary of Message Queue features: each feature is briefly described, the work required to use the feature is summarized, and references are provided to sections in this book that introduce these features and to the specific documents in the Message Queue documentation set that describe these features in detail.

Message Queue’s features, listed alphabetically in [Appendix B, “Message Queue Features”](#), can be roughly divided into the categories shown below.

- Integration Support
  - HTTP connections
  - Secure connections
  - C client support
  - SOAP support
  - JES Monitoring Framework support
  - J2EE resource adapters
- Security
  - Authentication
  - Authorization
  - JAAS-based authentication (see Authentication)
  - Encryption (see Secure connections)
- Scalability

- Thread management
- Broker clusters
- Queue delivery to multiple consumers
- Availability
  - Memory resource management
  - Message flow control to clients
  - Automatic reconnect
  - Reliable data persistence
  - Connection ping
  - Event Notification
- Manageability
  - Administration tools
  - Message-based monitoring API
  - JES Monitoring Framework support
  - Tunable performance
  - Configurable physical destinations
  - Broker configurations
  - Dead message queue
  - Client runtime logging
  - JMX-based administration
- Performance
  - Message compression
  - Tunable performance
  - Configurable physical destinations
- Flexible Server Configuration
  - Configurable persistence
  - LDAP server support
  - JNDI server support

# Feature List

TABLE B-1 Message Queue Features

Feature	Description and Reference
Administration tools	<p>The Message Queue service includes GUI and command line tools for managing destinations, transactions, durable subscriptions, administered object stores, user repositories, JDBC-compliant data stores, and server certificates.</p> <p><b>Reference</b></p> <p><a href="#">“Administration Tools” on page 65.</a></p> <p>Chapter 1, “Administrative Tasks and Tools,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Authentication	<p data-bbox="815 236 1229 288">Authenticate users seeking a connection to the broker.</p> <p data-bbox="815 309 1275 473">The Message Queue service allows users to connect to the broker by validating their name and password against values stored in a user repository. The repository can be a flat-file repository shipped with Message Queue or an LDAP repository (LDAP v2 or v3 protocol).</p> <p data-bbox="815 494 882 517"><b>To Use</b></p> <ol data-bbox="815 526 1222 644" style="list-style-type: none"> <li data-bbox="815 526 1222 578">1. Create a user repository or use the default instance.</li> <li data-bbox="815 586 1222 644">2. Use the <code>imqusermgr</code> tool to populate the repository.</li> </ol> <p data-bbox="815 664 1076 687"><b>JAAS-Based Authentication</b></p> <p data-bbox="815 708 1272 904">Application clients can also use authentication services based on the Java Authentication and Authorization Service (JAAS), which allows you to plug in a variety of services into the broker to authenticate Message Queue clients. The JAAS API is a core API in J2SE and therefore it is an integral part of Message Queue's runtime environment.</p> <p data-bbox="815 925 882 947"><b>To Use</b></p> <ol data-bbox="815 956 1275 1194" style="list-style-type: none"> <li data-bbox="815 956 1275 1043">1. The JAAS provider supplies a login module class that implements the authentication service.</li> <li data-bbox="815 1064 1275 1116">2. Obtain JAAS configuration file and specify its location using a system property.</li> <li data-bbox="815 1137 1275 1194">3. Configure broker properties that relate to JAAS support.</li> </ol> <p data-bbox="815 1215 911 1237"><b>Reference</b></p> <p data-bbox="815 1258 1243 1281">“<a href="#">Authentication and Authorization</a>” on page 61.</p> <p data-bbox="815 1302 1253 1352">Chapter 9, “Security,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Authorization	<p>Authorize users to perform specific operations.</p> <p>The Message Queue service allows you to create an access control properties file that specifies the operations users and groups of users can perform. The broker checks this file when a client seeks to create a connection, create a producer, create a consumer, or browse a queue.</p> <p><b>To Use</b></p> <p>Edit the access control properties file that is automatically created for the broker instance.</p> <p><b>Reference</b></p> <p><a href="#">“Authentication and Authorization” on page 61</a></p> <p>Chapter 9, “Security,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>
Automatic reconnect	<p>The administrator sets connection attributes on the connection factory administered object to enable automatic reconnection in the event of connection failure. Reconnection can be to the same broker or to another broker in a cluster if a cluster is used. You can specify how many times to try reconnection and the interval between attempts. For clustered brokers, you can also specify how often to iterate through a list of brokers and whether to iterate through the list in a specific order.</p> <p><b>Reference</b></p> <p><a href="#">“Connection Services” on page 55.</a></p> <p>Chapter 7, “Administered Objects,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Broker clusters	<p>The administrator can balance client connections and message delivery across a number of broker instances by grouping those instances into a broker cluster. The Message Queue service supports two kinds of clusters: conventional clusters and high availability clusters</p> <p><b>To Use Conventional Clusters</b></p> <ol style="list-style-type: none"> <li>1. Specify cluster configuration properties for each broker in the cluster. Specify properties that are the same for all brokers using a cluster configuration file.</li> <li>2. If there is a master broker, start the master broker</li> <li>3. Start the other brokers in the cluster.</li> </ol> <p><b>To Use High Availability Clusters</b></p> <ol style="list-style-type: none"> <li>1. Specify cluster configuration properties for each broker in the cluster (including JDBC-related properties). Specify properties that are the same for all brokers using a cluster configuration file.</li> <li>2. Install your JDBC driver's .jar file in the appropriate directory location.</li> <li>3. Use the <code>imqdbmgr</code> tool to create the database schema for the highly available data store.</li> <li>4. Start the brokers in the cluster.</li> </ol> <p><b>Reference</b></p> <p><a href="#">Chapter 4, “Broker Clusters,”</a></p> <p>Chapter 8, “Broker Clusters,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>
Broker configuration	<p>The administrator can set broker properties to tune Message Queue service performance. This includes routing services, persistence services, security, monitoring, and administered object management.</p> <p><b>Reference</b></p> <p><a href="#">Chapter 3, “Message Queue Service,”</a></p> <p>Chapter 4, “Broker Configuration,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
C client support	<p>C clients can use Message Queue messaging services to send and receive messages. The C API enables legacy C applications and C++ applications to participate in JMS-based messaging.</p> <p>Message Queue's C API is supported by a C client runtime that supports most of the standard JMS functionality, with the exception of the following: the use of administered objects; map, stream, or object message body types; distributed transactions; and queue browsers. The C client runtime also does not support most of Message Queue's enterprise features.</p> <p><b>Reference</b></p> <p>“Java and C Clients” on page 52.</p> <p><i>Sun Java System Message Queue 4.1 Developer's Guide for C Clients</i></p>
Client runtime logging	<p>Java clients can use all the J2SE 1.4 logging facilities to configure how the Message Queue client runtime outputs its logging information. Clients can choose to log the following events: changes in connection state and miscellaneous connection activities, session-related events, the creation of producers, consumers, and destinations, and the consumption and production of messages.</p> <p>Java clients can configure logging programmatically or by using configuration files.</p> <p><b>Reference</b></p> <p>“Client Runtime Logging” in <i>Sun Java System Message Queue 4.1 Developer's Guide for Java Clients</i></p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Compressed messages	<p>Java clients can set a message property to have the client runtime compress a message being sent. The runtime on the consumer side decompresses the message before it delivers it to the consumer. Additional properties are provided that you can use to determine whether compressing messages would actually improve performance.</p> <p><b>Reference</b></p> <p>“Message Body” on page 42.</p> <p>“Managing Message Size” in <i>Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients</i>.</p>
Configurable persistence	<p>The administrator can configure the broker to use the file-based persistent store provided with Message Queue or a JDBC-compliant database, such as Oracle 8i.</p> <p><b>To Use</b></p> <p>Set broker properties that relate to file-system persistent storage or JDBC-compliant storage.</p> <p><b>Reference</b></p> <p>“Persistence Services” on page 59.</p> <p>“Configuring a Persistent Data Store” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>



TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Configurable physical destinations	<p>The administrator can define some messaging behavior by setting physical destination properties when creating destinations. The following behavior can be configured for any destination: the maximum number of unconsumed messages or the maximum amount of memory allowed for such messages, which messages the broker should reject when memory limits are reached, the maximum number of producers and consumers, the maximum message size, the maximum number of messages delivered in a single batch, whether the destination can deliver only to local consumers, and whether dead messages on the destination can be moved to the dead message queue.</p> <p><b>Reference</b></p> <p>“Destinations and Routing Services” on page 56.</p> <p>Chapter 6, “Physical Destinations,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>
Connection ping	<p>The administrator can set a connection factory attribute to specify the frequency of a ping operation from the client runtime to the broker. This allows the client to preemptively detect a failed connection.</p> <p><b>Reference</b></p> <p>“Connection Services” on page 55.</p> <p>“Connection Services” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>
Dead message queue	<p>The Message Queue message service creates the dead message queue to hold messages that have expired or that the broker could not process. You can examine the contents of the queue to monitor, tune, or troubleshoot system performance.</p> <p><b>Reference</b></p> <p>“Destinations and Routing Services” on page 56.</p> <p>Chapter 6, “Physical Destinations,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Event Notification	<p>Java clients can listen for connection events (like closure or reconnection) and take appropriate action based on the notification type and the connection state.</p> <p><b>To Use</b></p> <ol style="list-style-type: none"> <li>1. Use the event notification API to create an event listener.</li> <li>2. Add code to the client application that will take appropriate action depending on the events captured by the event listener.</li> </ol> <p><b>Reference</b></p> <p>“Connection Event Notification” in <i>Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients</i></p>
HTTP connections	<p>Java clients can create HTTP connections to the broker.</p> <p>HTTP transport allows messages to be delivered through firewalls. Message Queue implements HTTP support using an HTTP tunnel servlet that runs in a web server environment. Messages produced by a client are wrapped by the client runtime as HTTP requests and delivered over HTTP through a firewall to the tunnel servlet. The tunnel servlet extracts the JMS message from the HTTP request and delivers the message over TCP/IP to the broker.</p> <p><b>To Use</b></p> <ol style="list-style-type: none"> <li>1. Deploy HTTP tunnel servlet on a web server.</li> <li>2. Configure broker’s <code>httpjms</code> connection service and start the broker.</li> <li>3. Configure HTTP connection.</li> <li>4. Obtain an HTTP connection to the broker. (Java clients only.)</li> </ol> <p><b>Reference</b></p> <p>“Connecting to the Broker” on page 25.</p> <p>Appendix C, “HTTP/HTTPS Support,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Interactive monitoring	<p>The administrator can use the <code>imqcmd metrics</code> command to monitor a broker remotely. Monitored data includes JVM metrics, broker message flow, connections, connection resources, messages, destination message flow, destination consumers, destination resource use.</p> <p><b>Reference</b></p> <p><a href="#">“Monitoring Services” on page 63.</a></p> <p>Chapter 10, “Monitoring Broker Operations,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p>
J2EE resource adapters	<p>Message Queue provides a resource adapter that can be plugged into a J2EE-compliant application server. By using Message Queue as a JMS provider, an application server meets the J2EE requirement that distributed components running in the application server be able to interact using reliable, asynchronous message.</p> <p><b>To Use</b></p> <p>Configure the adapter by setting adapter attributes.</p> <p><b>Reference</b></p> <p><a href="#">“J2EE Application Server Support” on page 85.</a></p> <p>Chapter 17, “JMS Resource Adapter Property Reference,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
JES Monitoring Framework support	<p>The JES Monitoring Framework allows administrators to use the same interface to manage any and all JES components. If you are using Message Queue with other JES components, it might be more convenient to manage these from a single console. Administrators can use the Sun Java System Monitoring Console to view performance statistics, create rules to monitor automatically, and acknowledge alarms. To enable JES monitoring, you must do the following:</p> <ul style="list-style-type: none"> <li>■ Install and configure the components in your deployment; for example, Message Queue and the Application Server.</li> <li>■ Enable and configure the Monitoring Framework for all your monitored components.</li> <li>■ Install the Monitoring Console on a separate host, start the master agent, and then start the web server.</li> </ul> <p>For information, see the <i>Sun Java Enterprise System Monitoring Guide</i>.</p>
JMX-Based Administration	<p>Java clients can use the JMX API to monitor and manage broker resources: the broker, services, connections, destinations, consumers, producers, and so on. You can use JMX-based management in different ways to monitor application performance, to monitor the broker, to automate tasks, or to write custom tools.</p> <p><b>Reference</b></p> <p><i>Sun Java System Message Queue 4.1 Developer's Guide for JMX Clients</i></p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
JNDI service provider support	<p>Clients can look up administered objects using the JNDI API.</p> <p>Administrators can use the <code>imqobjmgr</code> utility to add, list, update, and delete administered objects in an object store accessible using JNDI.</p> <p><b>Reference</b></p> <p><a href="#">“Administration Tools” on page 65</a></p> <p>Chapter 7, “Administered Objects,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p>
LDAP Server support	<p>The administrator can use LDAP servers for administered object store and for storing user information needed for authentication and authorization. By default Message Queue provides file-based storage for this data.</p> <p><b>To Use for Administered Objects</b></p> <ol style="list-style-type: none"> <li>1. Use the tools provided by the vendor to populate and manage the user repository.</li> <li>2. Set the LDAP-related broker properties.</li> <li>3. Set up access control for administrative users.</li> </ol> <p><b>Reference</b></p> <p><a href="#">“Security Services” on page 60.</a></p> <p>Chapter 9, “Security,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p> <p><b>To Use for User Repository</b></p> <ol style="list-style-type: none"> <li>1. Use the tools provided by the vendor to set up the LDAP server.</li> <li>2. Set the LDAP-related broker properties to define the initial context and the location of the store.</li> <li>3. Set the LDAP-related broker properties that relate to securing the LDAP server operations.</li> </ol> <p><b>Reference</b></p> <p>Chapter 9, “Security,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Memory resource management	<p>The administrator can configure the following behavior:</p> <ol style="list-style-type: none"> <li>1. Set properties on a destination to specify the maximum number of producers, the maximum number of messages, and the maximum size of any one message.</li> <li>2. Set properties on a destination to control message flow.</li> <li>3. Set properties on a destination to manage message flow for each destination.</li> <li>4. Set properties on the broker to specify message limits on all destinations for that broker.</li> <li>5. Set properties on the broker to specify thresholds of available system memory at which the broker takes action to prevent memory overload. The action taken depends on the state of memory resources.</li> </ol> <p><b>Reference</b></p> <p>“Destinations and Routing Services” on page 56.</p> <p>Chapter 4, “Broker Configuration,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p>
Message compression	<p>The developer can set a message header property to have the client runtime compress a message before sending it. The client runtime on the consumer side decompresses the message before delivering it to the consumer.</p> <p><b>Reference</b></p> <p>“Message Properties” on page 41.</p> <p>“Message Compression” in <i>Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients</i></p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Message flow control to clients	<p>The administrator or the developer can configure a connection to specify various flow limits and metering schemes to minimize the collision of payload and control messages, and thereby to maximize message throughput.</p> <p><b>To Use</b></p> <p>Set the flow-control attributes for the connection factory administered object (administrator), or set the flow-control properties for the connection factory (developer).</p> <p><b>Reference</b></p> <p><a href="#">“Connection Factories and Connections” on page 38.</a></p> <p>“Connection Services” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p> <p>“Connection Factory Attributes” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p>
Message-based monitoring API	<p>Java clients can use a monitoring API to create custom monitoring applications. A monitoring application is a consumer that retrieves metrics messages from special metrics topic destinations.</p> <p><b>To Use</b></p> <ol style="list-style-type: none"> <li>1. Write a metrics monitoring client.</li> <li>2. Set broker properties to configure the broker’s metrics message producer.</li> <li>3. Set access controls on metrics topic destinations.</li> <li>4. Start the monitoring client.</li> </ol> <p><b>Reference</b></p> <p><a href="#">“Monitoring Services” on page 63.</a></p> <p>Chapter 4, “Using the Metrics Monitoring API,” in <i>Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients</i>.</p> <p>Chapter 10, “Monitoring Broker Operations,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Queue delivery to multiple consumers	<p>Clients can register more than one consumer for a given queue.</p> <p>The administrator can specify the maximum number of active consumers and the maximum number of backup consumers for the queue. The broker distributes messages to the registered consumers, balancing the load among them in order to allow the system to scale.</p> <p><b>To Use</b></p> <p>Set physical destination properties <code>maxNumActiveConsumers</code> and <code>maxNumBackupConsumers</code>.</p> <p><b>Reference</b></p> <p>“Point-To-Point Messaging” on page 32.</p> <p>Chapter 15, “Physical Destination Property Reference,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>
Reliable data persistence	<p>To obtain absolute reliability you can require that the operating system write the data synchronously to the persistent store by setting the <code>imq.persist.file.sync.enabled</code> property to <code>true</code>. This eliminates possible data loss due to system crashes, but at the expense of performance. Note that although the data is not lost, it is not available to any other broker (in a cluster) because data is not currently shared by clustered brokers. When the system comes back up, the broker can reliably resume operations.</p> <p><b>Reference</b></p> <p>“Persistence Services” on page 59.</p> <p>“Persistence Properties” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p> <p>.</p>



TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Secure connections	<p>Clients can secure transmission of messages using the Secure Socket Layer (SSL) standard over TCP/IP and HTTP transports. These SSL-based connection services allow for the encryption of messages sent between clients and broker.</p> <p>SSL support is based on self-signed server certificates. Message Queue provides a utility that generates a private/public key pair and embeds the public key in a self-signed certificate. This certificate is passed to any client requesting a connection to the broker, and the client uses the certificate to set up an encrypted connection.</p> <p><b>To Use</b></p> <ol style="list-style-type: none"> <li>1. Generate a self-signed or signed certificate.</li> <li>2. Enable the secure service.</li> <li>3. Start the broker.</li> <li>4. Configure client security connection properties and run the client.</li> </ol> <p><b>Reference</b></p> <p><a href="#">“Connecting to the Broker” on page 25.</a></p> <p><a href="#">“Security Services” on page 60.</a></p> <p>Chapter 9, “Security,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p> <p>“Working With Secure Connections” in <i>Sun Java System Message Queue 4.1 Developer’s Guide for C Clients</i></p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
SOAP support	<p>Clients can receive SOAP (XML) messages and they can wrap them as JMS messages and use Message Queue to exchange them as they would a JMS message.</p> <p>Clients can use a special servlet to receive SOAP messages; they can use a utility class to wrap a SOAP message as a JMS message; they can use another utility class to extract the SOAP message from the JMS message. Clients can use standard SAAJ libraries to assemble and disassemble a SOAP message.</p> <p><b>Reference</b></p> <p><a href="#">“Working with SOAP Messages” on page 51.</a></p> <p>“Working With SOAP Message,” Chapter 5, “Working with SOAP Messages,” in <i>Sun Java System Message Queue 4.1 Developer’s Guide for Java Clients</i>.</p>
Thread management	<p>The administrator can specify the maximum and minimum number of threads assigned to any specific connection service. The administrator can also determine whether a connection service could increase throughput by using a shared thread model, which allows threads dedicated to idle connections to be used by other connections.</p> <p><b>To Use</b></p> <p>Set connection service thread-related properties.</p> <p><b>Reference</b></p> <p><a href="#">“Thread Pool Management” on page 56.</a></p> <p>Chapter 4, “Broker Configuration,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i>.</p>

TABLE B-1 Message Queue Features (Continued)

Feature	Description and Reference
Tunable performance	<p>The administrator can set broker properties to adjust memory usage, threading resources, message flow, connection services, reliability parameters, and other elements that affect message throughput and system performance.</p> <p><b>Reference</b></p> <p>“Monitoring Services” on page 63.</p> <p>“Monitoring Services” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p> <p>Chapter 11, “Analyzing and Tuning a Message Service,” in <i>Sun Java System Message Queue 4.1 Administration Guide</i></p>



# Glossary

---

This glossary provides information about terms and concepts you might encounter while using Message Queue.

<b>acknowledgement</b>	Control messages exchanged between clients and broker to ensure reliable delivery. There are two general types of acknowledgement: client acknowledgements and broker acknowledgements.
<b>administered objects</b>	A pre-configured object—a connection factory or a destination—that encapsulates provider-specific implementation details, and is created by an administrator for use by one or more JMS clients. The use of administered objects allows JMS clients to be provider-independent. Administered objects are placed in a JNDI name space by and are accessed by JMS clients using JNDI lookups.
<b>asynchronous messaging</b>	An exchange of messages in which the sending of a message does not depend upon the readiness of the consumer to receive it. In other words, the sender of a message need not wait for the sending method to return before it continues with other work. If a message consumer is busy or offline, the message is sent and subsequently received when the consumer is ready.
<b>authentication</b>	The process by which only verified users are allowed to set up a connection to a broker.
<b>authorization</b>	The process by which a message service determines whether a user can access message service resources, such as connection services or destinations, to perform specific operations supported by the message service.
<b>broker</b>	The Message Queue entity that manages message routing, delivery, persistence, security, and logging, and that provides an interface for monitoring and tuning performance and resource use.
<b>client</b>	An application (or software component) that interacts with other clients using a message service to exchange messages. The client can be a producing client, a consuming client, or both.
<b>client identifier</b>	An identifier that associates a connection and its objects with a state maintained by the Message Queue broker on behalf of the client.
<b>client runtime</b>	Message Queue software that provides messaging clients with an interface to the Message Queue message service. The client runtime supports all operations needed for clients to send messages to destinations and to receive messages from destinations.
<b>cluster</b>	Two or more interconnected brokers that work in concert to provide scalable messaging services. In the event of failover and reconnection, conventional clusters provide service availability; high availability clusters provide service and data availability.

<b>cluster connection service</b>	A private protocol that enables brokers in a cluster to provide reliable, synchronized service.
<b>connection</b>	A communication channel between a client and a broker used to pass both payload messages and control messages.
<b>connection factory</b>	The administered object the client uses to create a connection to a broker. This can be a <code>ConnectionFactory</code> object, a <code>QueueConnectionFactory</code> object or a <code>TopicConnectionFactory</code> object.
<b>consumer</b>	An object ( <code>MessageConsumer</code> ) created by a session that is used for receiving messages sent from a destination. In the point-to-point delivery model, the consumer is a receiver or browser ( <code>QueueReceiver</code> or <code>QueueBrowser</code> ); in the publish/subscribe delivery model, the consumer is a subscriber ( <code>TopicSubscriber</code> ).
<b>data store</b>	A database where information (durable subscriptions, data about destinations, persistent messages, auditing data) needed by the broker is permanently stored.
<b>dead message</b>	A message that is removed from the system for a reason other than normal processing or explicit administrator action. A message might be considered dead because it has expired, because it has been removed from a destination due to memory limit overruns, or because of failed delivery attempts. You can choose to store dead messages on the dead message queue.
<b>dead message queue</b>	A specialized destination created automatically at broker startup that is used to store dead messages for diagnostic purposes.
<b>delivery mode</b>	An indicator of the reliability of messaging: whether messages are guaranteed to be delivered and successfully consumed once and only once (persistent delivery mode) or guaranteed to be delivered at most once (non-persistent delivery mode).
<b>delivery model</b>	The model by which messages are delivered: either point-to-point or publish/subscribe. In JMS there are separate programming domains for each, using specific client runtime objects and specific destination types (queue or topic), as well as a unified programming domain.
<b>destination</b>	The physical destination in a Message Queue broker to which produced messages are delivered for routing and subsequent delivery to consumers. This physical destination is identified and encapsulated by an administered object that a client uses to specify the destination for which it is producing messages and/or from which it is consuming messages.
<b>domain</b>	A set of objects used by JMS clients to program JMS messaging operations. There are two programming domains: one for the point-to-point delivery model and one for the publish/subscribe delivery model.
<b>encryption</b>	A mechanism for protecting messages from being tampered with during delivery over a connection.
<b>group</b>	The group to which the user of a Message Queue client belongs for purposes of authorizing access to connections, destinations, and specific operations.
<b>JMS provider</b>	A product that implements the JMS interfaces for a messaging system and adds the administrative and control functions needed to configure and manage that system.

---

<b>message service</b>	A middleware service that provides asynchronous, reliable exchange of messages between distributed components or applications. It includes a broker, the client runtime, the several data stores needed by the broker to carry out its functions, and the administrative tools needed to configure and monitor the broker and to tune performance.
<b>messages</b>	Asynchronous requests, reports, or events that are consumed by messaging clients. A message has a header (to which additional fields can be added) and a body. The message header specifies standard fields and optional properties. The message body contains the data that is being transmitted.
<b>messaging</b>	A system of asynchronous requests, reports, or events used by enterprise applications that allows loosely coupled applications to transfer information reliably and securely.
<b>producer</b>	An object ( <code>MessageProducer</code> ) created by a session that is used for sending messages to a destination. In the point-to-point delivery model, a producer is a sender ( <code>QueueSender</code> ); in the publish/subscribe delivery model, a producer is a publisher ( <code>TopicPublisher</code> ).
<b>queue</b>	An object created by an administrator to implement the point-to-point delivery model. A queue is always available to hold messages even when the client that consumes its messages is inactive. A queue is used as an intermediary holding place between producers and consumers.
<b>selector</b>	A message header property used to sort and route messages. A message service performs message filtering and routing based on criteria placed in message selectors.
<b>session</b>	A single threaded context for sending and receiving messages. This can be a queue session or a topic session.
<b>topic</b>	An object created by an administrator to implement the publish/subscribe delivery model. A topic may be viewed as node in a content hierarchy that is responsible for gathering and distributing messages addressed to it. By using a topic as an intermediary, message publishers are kept separate from message subscribers.
<b>transaction</b>	An atomic unit of work that must either be completed or entirely rolled back.





# Index

---

## A

- access control, 62
- access control file, 60
- admin-created destinations, 57
- administered objects
  - introduced, 22
  - managing, 67
  - use of, 23
- administration tools, 28
- application servers, and Message Queue, 85-86
- authentication
  - about, 61-62
  - components needed for, 61
  - JAAS-based, 62
  - use and reference, 92
- authorization
  - See also* access control file
  - about, 61-62
  - components needed for, 61
  - use and reference, 93
- AUTO\_ACKNOWLEDGE mode, 46
- auto-created destinations, 57, 77

## B

- broker acknowledgements
  - message consumption, and, 46
  - suppression of, 39
- broker clusters
  - architecture of conventional, 72
  - architecture of high availability, 73

## broker clusters (*Continued*)

- cluster configuration file, 81
- cluster configuration properties, 81
- comparing models for, 82
- configuration change record, 81
- connection service for, 72
- conventional, 71
- destination handling in, 79
- high availability, 71
- master broker, 81, 82
- message delivery in, 74
- message traffic in, 75
- models for, 71
- producing to auto-created destinations, 77
- propagation of information in, 81
- replication of topic destination in, 78
- request-reply implementation of, 76
- scope of destinations in, 75
- synchronization of information in, 81
- use and reference, 94

## brokers

- administration of, 68
- automatic reconnection to, 39, 93
- connecting to, 26
- development environment, 68
- firewalls, connecting through, 55
- GUI-based administration of, 67
- interconnected
  - See* broker clusters
- introduced, 26
- JMX, and, 67
- limit behaviors, 59

brokers (*Continued*)

- logging
    - See* logger
  - maintenance, 69
  - master broker, 81, 82
  - memory management, 58, 59
  - metrics
    - See* broker metrics
  - monitoring, 99
  - monitoring APIs, 103
  - performance of, tuning, 107
  - production environment, 68
  - programmatic management of, 67
  - properties, 55
  - recovery from failure, 59
  - restarting, 59
  - services used by, 55
  - starting, 66
  - tools for administering, 65
  - windows service, as, 67
- built-in persistence, 60
- BytesMessage type, 42

**C**

- C clients, 27, 52, 95
- CLIENT\_ACKNOWLEDGE mode, 46
- client acknowledgements, 46
- client authentication, 38
- clients
  - C and C++, 27, 52, 95
  - Java, 27, 52
  - runtime support for, 27
- cluster configuration file, 81
- cluster configuration properties, 81
- components
  - EJB, 83
  - MDB, 84
- connection factory administered objects
  - as JMS programming object, 38
  - defined, 22
- connection objects, 38
- connection services, 26
  - about, 54

connection services (*Continued*)

- automatic reconnection, 93
  - configuring, 55
  - HTTP support, 98
  - managing, 67
  - message flow, 103
  - pinging service, 97
  - port mapper
    - See* port mapper
  - secure, 105
  - thread management, 106
- consumers
- as JMS clients, 21
  - as JMS programming object, 43
  - asynchronous, 43
  - delivery to, 43
  - durable, 39
  - load balancing consumption, 34
  - multiple for a queue, 104
  - synchronous, 43
- containers
- EJB, 84
  - MDB, 84
- control messages, 49

**D**

- data store, 48
  - about, 59
  - flat-file, 60
  - JDBC-accessible, 60
- dead message queue
  - about, 57
  - use and reference, 97
- delivery, reliable, *See* reliable delivery
- delivery mode, 40
- design and performance, 32
- destination administered objects
  - as JMS programming object, 43
  - defined, 22
- destinations
  - configuring, 58
  - creating, 39
  - handling of in clusters, 79

destinations (*Continued*)

- kinds of, 57
- limits for, 58
- managing, 57
- scope of in clusters, 75
- temporary, 39, 44

## distributed transactions

- See also* XA connection factories
- about, 47
- JMS requirements, and, 88
- XA resource manager, 48

DUPS\_OK\_ACKNOWLEDGE mode, 47

durable subscriptions, 48

**E**

EJB containers, 84

encryption, 63

**F**

firewalls, 55, 56

**H**

HTTP connections, 98

**I**

imqbrokerd utility, 66

imqcmd utility, 67

imqdbmgr utility, 67

imqkeytool utility, 67

imqobjmgr utility, 67

imqsvcadm utility, 67

imqusermgr utility, 61, 67

**J**

## J2EE applications

- EJB specification, 83
- JMS, and, 20, 83
- message-driven beans
  - See* message-driven beans
- Message Queue and, 30

J2EE resource adapters, 99

JAAS-based authentication, 62

Java clients, 27, 52

## JDBC support

- about, 60
- managing, 67

JES Monitoring Framework, 64, 100

## JMS

- domains and APIs, 36
- message properties, standard, 41
- messaging objects, 21
- messaging patterns, 21
- optional features in Message Queue, 87
- provider, 20
- reserved properties, 88
- runtime support for, 27
- specification, 20

JMS applications, 37

JMS clients, 52

JMSCorrelationID message header field, 40

JMSDeliveryMode message header field, 41

JMSDeliveryMode message header field, 40

JMSDestination message header field, 40

JMSExpiration message header field, 41

JMSExpiration message header field, 40

JMSMessageID, 87

JMSMessageID message header field, 40

JMSPriority message header field, 41

JMSPriority message header field, 40

JMSRedelivered message header field, 40

JMSReplyTo message header field, 43

JMSReplyTo message header field, 40

JMSTimestamp message header field, 40

JMSType message header field, 40

JMX-based administration, 67

JNDI support, 101

**L**

- LDAP repository, 61
- LDAP server support, 101
- listeners
  - as JMS programming object, 43
  - MDBs, and, 84
  - serializing, 40
- logger
  - about, 64
  - output channels, 64
- logging, *See* logger

**M**

- MapMessage type, 42
- master broker, 81, 82
- MDB, *See* message-driven beans
- MDB containers, 84
- memory management, 58, 102
- message consumers, *See* consumers
- message-driven beans
  - about, 84
  - application server support, 85-86
  - deployment descriptor, 84
  - MDB container, 84
- message header fields
  - JMS message, 40-41
  - overriding, 39, 87
- message listeners, *See* listeners
- message-oriented middleware, 15, 16
- message producers, *See* producers
- Message Queue
  - application servers, and, 85-86
  - development environment, 68
  - features summary, 89
  - JMS optional features, 87
  - production environment, 68
- message service
  - administration, 28
  - components of, 53
  - introduced, 24
  - memory management, 102
  - scaling, 28
- Message type, 42

- messages
  - body of, 42
  - body types, 42
  - broadcasting, 36
  - compressing, 96, 102
  - compression of, 42
  - consumption of, 43
  - control, 49
  - correspondence, establishing, 40
  - delivery mode, 40
  - destination of, 40
  - expiration of, 40
  - headers
    - See* message header fields
  - ID of, 40
  - JMS, 40
  - JMS properties of, 41-42
  - JMSReplyTo header field, 44
  - listeners for, 43
  - load balancing consumption of, 34
  - payload, 48
  - persistent, 41
  - priority of, 40
  - processing of, 49
  - producing and consuming, 38
  - properties, 41
  - publishing, 34
  - redelivery flag, 40
  - reliable delivery of, 46
  - reply-to destination, 40
  - selecting, 40, 43
  - SOAP, 51
  - storage of, 48
  - timestamp for, 40
- messaging domains
  - APIs and, 36
  - introduced, 32
  - point-to-point, 32
  - publish/subscribe, 34
- messaging provider, 18
- metrics
  - data
    - See* broker metrics
  - message producer, 64

metrics (*Continued*)  
 messages, 64  
 reports, 63  
 middleware, 15, 16  
 monitoring, support for JES, 64, 100  
 monitoring APIs, 103  
 monitoring services, 55

## O

object request broker, 16  
 ObjectMessage type, 42

## P

payload messages, 48  
 performance, 107  
 performance and design, 32  
 permissions  
   access control properties file, 62  
   Message Queue operations, 62  
 persistence  
   built-in, 60  
   configurable, 96  
   data, of, 104  
   plugged-in  
     *See* plugged-in persistence  
 persistence services, 54  
 persistent data store, 48  
 physical destinations  
   configuring, 58  
   creating, 39  
   kinds of, 57  
   limits for, 58  
   managing, 57, 67  
   temporary, 39, 44  
 plugged-in persistence, 60  
 point-to-point messaging, 32  
 port mapper, 56  
 ports, dynamic allocation of, 56  
 producers  
   as JMS clients, 21  
   as JMS programming object, 42

producers (*Continued*)  
 creating, 42  
 publish/subscribe messaging, 34  
 publishing, 34

## Q

queue browser, 34, 39, 40  
 queues, 39

## R

reliable delivery  
   data persistence, 104  
   JMS specification, 46-48  
 request-reply pattern, 44, 76  
 resource adapters, 30, 85, 99  
 routing services, 54

## S

Secure Socket Layer standard, *See* SSL  
 security, 60, 105  
 security services, 55  
 selectors, 43  
 self-signed certificates, 67  
 sessions  
   as JMS programming object, 39  
   JMS client acknowledgements, 46  
   threading and, 39  
   transacted, 46  
 SOAP messages, 51  
 SOAP support, 27, 106  
 SSL  
   about, 63  
   feature description, 105  
   self-signed certificates, 67  
 St reamMessage type, 42  
 subscribers  
   durable, 35, 44, 48  
   introduced, 34

## T

- temporary destinations, 44, 57
- TextMessage type, 42
- thread management, 106
- threading model, 56
- timestamps, 40
- TLS protocol, 61
- topic destinations, 78
- topics, 39
- transactions
  - distributed
    - See* distributed transactions
  - processing of, 47

## U

- unified APIs, 36
- user data, 61
- users, managing, 67

## X

- XA resource manager, *See* distributed transactions