Sun Java™ System

# Message Queue 3 Technical Overview

2005Q1

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

Part No: 819-0069-10

# Contents

# List of Figures

# List of Tables

# Preface

This book, the Sun Java™ System Message Queue 3 2005Q1 *Technical Overview*, provides an introduction to the technology, concepts, architecture, capabilities, and features of the Message Queue messaging service.

As such, the *Message Queue Technical Overview* provides the foundation for other books within the Message Queue documentation set. You should read this book before reading the other books in the Message Queue documentation set.

This preface contains the following sections:

# Who Should Use This Book

This guide is meant for administrators, application developers, 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.

A Message Queue administrator is responsible for setting up and managing a Message Queue messaging system, in particular the Message Queue message server at the heart of this system. This book does not assume any knowledge or understanding of messaging systems.

An application developer is responsible for writing Message Queue client applications that use the Message Queue service to exchange messages with other client applications. 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 1** Book Contents and Organization

| Chapter | Description |
|---|---|
| Chapter 1, “Conceptual Foundations” | Provides the conceptual background to Message Queue, describing enterprise messaging systems and introducing Java Message Service concepts and terminology |
| Chapter 2, “Introduction to Message Queue” | Introduces the Message Queue service by discussing its architecture and describing its enterprise-strength features and capabilities |
| Chapter 3, “Reliable Message Delivery” | Describes how the Message Queue service provides reliable message delivery for messaging applications |
| Chapter 4, “Message Server” | Discusses the internal structure of the broker, describing the various broker components and their functions. Describes the different approaches to using Message Queue in development and production environments. |
| Chapter 5, “Broker Clusters” | Discusses the architecture and internal functioning of Message Queue broker clusters |
| Chapter 6, “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 |
| Glossary | Provides information about terms and concepts you might encounter while using Message Queue |

# Conventions Used in this Book

This section provides information about the conventions used in this document.

## Text Conventions

**Table 2** Document Conventions

| Format | Description |
|---|---|
| *italics* | Italicized text represents a placeholder. Substitute an appropriate clause or value where you see italic text. Italicized text is also used to designate a document title, for emphasis, or for a word or phrase being introduced. |
| `monospace` | Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs. |
| `[]` | Square brackets to indicate optional values in a command line syntax statement. |
| ALL CAPS | Text in all capitals represents file system types (GIF, TXT, HTML and so forth), environment variables (IMQ_HOME), or acronyms (Message Queue, JSP). |
| Key+Key | Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously. |
| Key-Key | Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key. |

# Directory Variable Conventions

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

**Table 3** Message Queue Directory Variables

| Variable | Description |
|---|---|
| **IMQ_HOME** | This is generally used in Message Queue documentation to refer to the Message Queue base directory (root installation directory):<br>• On Solaris, there is no root Message Queue installation directory. Therefore, IMQ_HOME is not used in Message Queue documentation to refer to file locations on Solaris.<br>• On Solaris, for Sun Java System Application Server the root Message Queue installation directory is /imq under the Application Server base directory.<br>• On Windows, the root Message Queue installation directory is set by the Message Queue installer (by default, as C:\Program Files\Sun\MessageQueue3).<br>• On Windows, for Sun Java System Application Server, the root Message Queue installation directory is /imq under the Application Server base directory.<br>• On Linux, there is no root Message Queue installation directory. Therefore, IMQ_HOME is not used in Message Queue documentation to refer to file locations on Linux. |
| **IMQ_VARHOME** | This is the /var 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.<br>• On Solaris, IMQ_VARHOME defaults to the /var/imq directory.<br>• On Solaris, for Sun Java System Application Server, Evaluation Edition, IMQ_VARHOME defaults to the IMQ_HOME/var directory.<br>• On Windows IMQ_VARHOME defaults to the IMQ_HOME\var directory.<br>• On Windows, for Sun Java System Application Server, IMQ_VARHOME defaults to the IMQ_HOME\var directory.<br>• On Linux, IMQ_VARHOME defaults to the /var/opt/imq directory |

**Table 3** Message Queue Directory Variables *(Continued)*

| Variable | Description |
|---|---|
| **IMQ_JAVAHOME** | This is an environment variable that points to the location of the Java™ runtime (JRE) required by Message Queue executables:<br>• On Solaris, IMQ_JAVAHOME looks for the java runtime in the following order, but a user can optionally set the value to wherever the required JRE resides.<br>Solaris 8 or 9:<br>/usr/jdk/entsys-j2se<br>/usr/jdk/jdk1.5.*<br>/usr/jdk/j2sdk1.5.*<br>/usr/j2se<br>Solaris 10:<br>/usr/jdk/entsys-j2se<br>/usr/java<br>/usr/j2se<br>• On Linux, Message Queue first looks for the java runtime in the following order, but a user can optionally set the value of IMQ_JAVAHOME to wherever the required JRE resides.<br>/usr/jdk/entsys-j2se<br>/usr/java/jre1.5.*<br>/usr/java/jdk1.5.*<br>/usr/java/jre1.4.2*<br>/usr/java/j2sdk1.4.2*<br>• On Windows, IMQ_JAVAHOME defaults to IMQ_HOME\jre, but a user can optionally set the value to wherever the required JRE resides. |

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 (/).

# Related Documentation

In addition to this guide, Message Queue provides additional documentation resources.

## The Message Queue Documentation Set

The documents that comprise the Message Queue documentation set are listed in Table 4 in the order in which you would normally use them.

**Table 4** Message Queue Documentation Set

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

# Online Help

Message Queue includes command line utilities for performing Message Queue message service administration tasks. To access the online help for these utilities, see the *Message Queue Administration Guide*.

Message Queue also includes a graphical user interface (GUI) administration tool, the Administration Console (`imqadmin`). Context sensitive online help is included in the Administration Console.

# JavaDoc

Message Queue Java client API (including the JMS 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 such as Netscape or Internet Explorer. It includes standard JMS API documentation as well as Message Queue-specific APIs for Message Queue administered objects (see Chapter 3 of the *Message Queue Developer's Guide for Java Clients*), which are of value to developers of messaging applications.

# Example Client Applications

A number of example applications that provide sample client application code are included in a directory that depends upon the operating system (see the *Message Queue Administration Guide*).

See the README file located in that directory and in each of its subdirectories.

## 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.

# Related 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 by or in connection with the 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 document title and part number. The part number is a seven-digit or nine-digit number that can be found on the title page of the book or at the top of the document.

Chapter 1

# Conceptual Foundations

Sun Java™ System Message Queue (Message Queue) provides reliable, *asynchronous messaging* that can integrate distributed applications and components across an enterprise. Processes running on different platforms and operating systems can connect to the service to interact with each other.

Message Queue is a standards-based *messaging* solution that implements the Java™ Message Service (JMS) open standard. In addition, Message Queue provides the interoperability, security, scalability, availability, manageability, and other features required by large-scale enterprise deployments.

This chapter provides the conceptual foundation for Message Queue. It covers the following topics:



If you are already familiar with JMS concepts and terminology, you can skip to Chapter 2, "Introduction to Message Queue."

# Enterprise Messaging Systems

Enterprise messaging systems enable independent distributed applications or application components to interact through *messages*. These components, whether on the same host, the same network, or loosely connected through the Internet, use messaging to pass data and to coordinate their respective functions.

For large numbers of components to be able to exchange messages simultaneously and to support high density throughputs, the sending of a message cannot depend upon the readiness of the consumer to receive it. If a message consumer is busy or offline, the system must allow for a message to be received when the consumer is ready. This de-coupling of the sending and receiving of a message is known as asynchronous message delivery.

The asynchronous messaging model lends itself extremely well to the task of integrating complex systems, where it is neither feasible nor desirable for one component to hold up another in the process of doing work. While asynchronous messaging gives up some of the control that synchronous systems allow, it adds great flexibility to the interplay of components. It also adds robustness, inasmuch as the failure of one component does not translate into the failure of the whole.

## Requirements of Enterprise Messaging Systems

Enterprise application systems typically consist of large numbers of distributed components exchanging many thousands of messages in round-the-clock, mission-critical operations. To support such systems, in addition to supporting asynchronous messaging, an enterprise messaging system must meet the following requirements:

**Reliable delivery.** Messages from one component to another must not be lost due to network or system failure. This means the system must be able to guarantee message delivery.

**Security.** The messaging system must support basic security features: authentication of users, authorized access to messages and resources, and over-the-wire encryption.

**Scalability.** The messaging system must be able to accommodate increasing loads—increasing numbers of users and increasing numbers of messages—without a substantial loss of performance or message throughput. As businesses and applications expand, this becomes an important requirement.

**Availability.** The messaging system must function with very little down time. This means that when failures occur, the system contains enough redundancy to continue to provide messaging services.

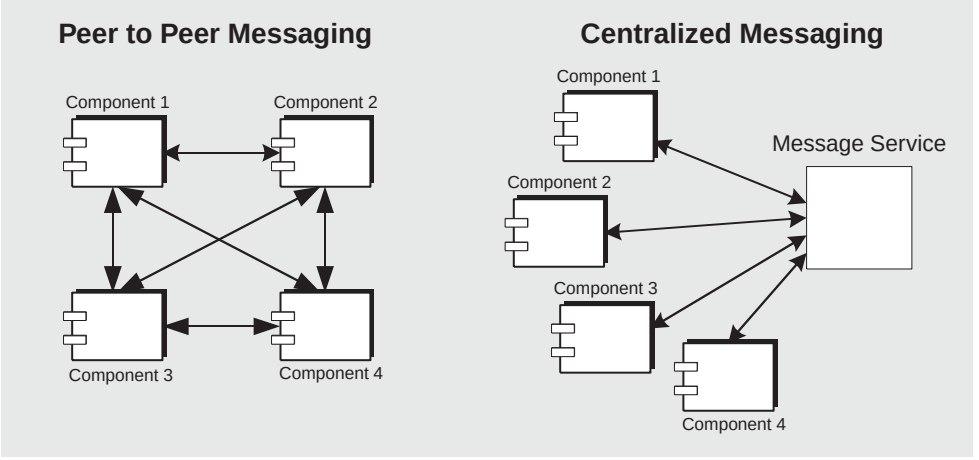
**Manageability.** The messaging system must provide tools for monitoring and managing message delivery. An administrator must be able to optimize system resources and to tune system performance.

# Centralized (MOM) Messaging

Message Queue uses a centralized messaging system, as shown in Figure 1-1. In such a system, each messaging component maintains a connection to one central message service. Components interact with the message service through a well-defined interface.

An alternative peer-to-peer system, in which every messaging component maintains a connection to every other component, is illustrated in the left of the figure. A peer-to-peer system allows for fast, secure, and reliable delivery; however, the code for supporting reliability and security must reside in each component. The sending and receiving of a message are closely coupled, making asynchronous delivery hard to achieve. As components are added to the system, the number of connections rises geometrically, so the system scales poorly. Centralized management is also problematic in a peer-to-peer system.

**Figure 1-1** Centralized vs. Peer-to-peer Messaging



In the centralized system, the preferred approach for enterprise messaging, the message service provides for routing and delivery of messages between components, and is responsible for reliable delivery and security. Because components in this system are loosely coupled, asynchronous messaging is easier to achieve.

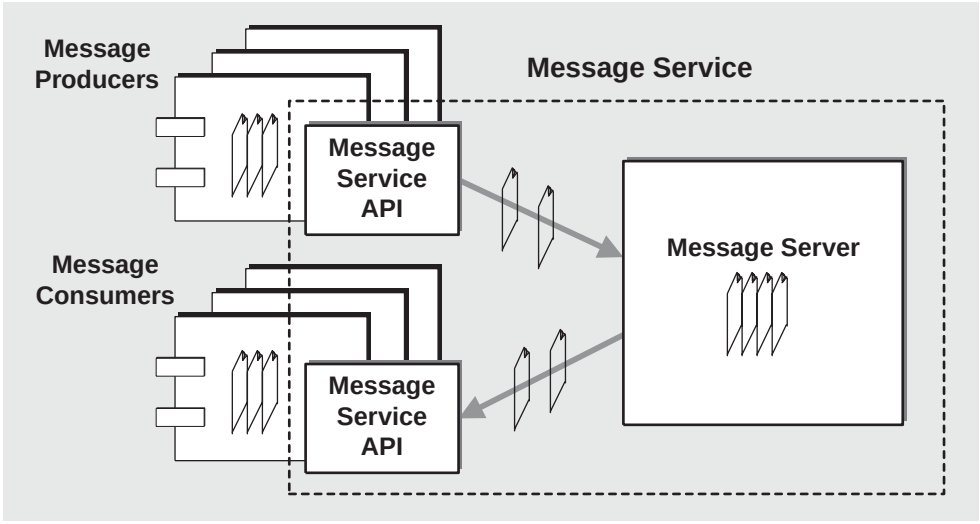
As messaging components are added to the system, the number of connections rises linearly, making it easy to scale the system by scaling the message service. In addition to connecting messaging clients, a central message service also provides an administrative interface that can be used to configure behavior, monitor performance, and tune the service to satisfy the needs of each messaging client.

## Basic Message Service Architecture

The basic architecture of a centralized messaging system is illustrated in Figure 1-2. It consists of message *producers* and message *consumers* that exchange messages by way of a common *message service*. Any number of message producers and consumers can reside in the same messaging component (or application).

A message producer uses the message service programming API to send a message to the *message server*. The message server routes and delivers the message to one or more message consumers that have registered an interest in the message. A consumer uses the message service programming API to receive messages. The message service is responsible for guaranteeing delivery of the message to all appropriate consumers.

**Figure 1-2** Message Service Architecture




Perhaps the best metaphor for this process is that of exchanging mail: Although a piece of mail is addressed to its eventual receiver, the mail is routed through the post office, resting in several intermediate locations before its recipient retrieves it from the mailbox.

# Java Message Service (JMS) Basics

Message Queue is an enterprise messaging system that implements the Java Message Service (JMS) open standard: it is a *JMS provider*. JMS concepts are therefore fundamental to understanding how the Message Queue service works.

The JMS specification prescribes a set of rules and semantics that govern reliable, asynchronous messaging. The specification defines a message structure, a programming model, and an API.

This section explains JMS concepts and terminology needed to understand the remaining chapters of this book. It covers the following topics:



## JMS Message Structure

In Message Queue data is exchanged using JMS messages. According to the JMS specification, a message, created by a producing *client*, is composed of three parts: a header, properties, and a body.

### *Header*

A header is required of every JMS message. Header fields contain values used for routing and identifying messages.

The header values can be set in a number of ways:

- by the JMS provider, automatically, during the process of producing or delivering the message
- by the producing client, through settings specified when the message producers are created
- by the producing client, on a message by message basis

For information about the header fields defined by JMS, see the *Message Queue Developer's Guide for Java Clients* or the *Message Queue Developer's Guide for C Clients*. These header fields allow you to define the destination of the message, the time of expiration, its priority, and so on.

### *Properties*

A message can include optional header fields, called *properties*. They are specified as property name and property value pairs. Properties, which can be thought of as extensions of the message header, might include information about which process created the data, the time it was created, and the structure of each piece of data. The JMS provider might also add properties that affect the processing of the message, such as whether it should be compressed or how it should be discarded at end of life.

The JMS provider can use message properties as *selectors* to sort and route messages. A producing client can place application-specific properties in the message, and a consuming client can choose to receive only messages whose properties have particular values. For instance, a consuming client might indicate an interest only for payroll messages concerning part-time employees located in New Jersey. Messages that do not meet the specified selection criteria are not delivered to the client.

Selectors simplify the work of consuming clients and eliminate the overhead of delivering messages to clients that don't want them. However, they add some overhead to the message service, which has to process the selection criteria. Message selector syntax and semantics are outlined in the JMS specification.

### *Message Body Types*

The type of a JMS message determines the contents of its body, as specified in Table 1-1.

**Table 1-1** 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. |

# JMS Programming Model

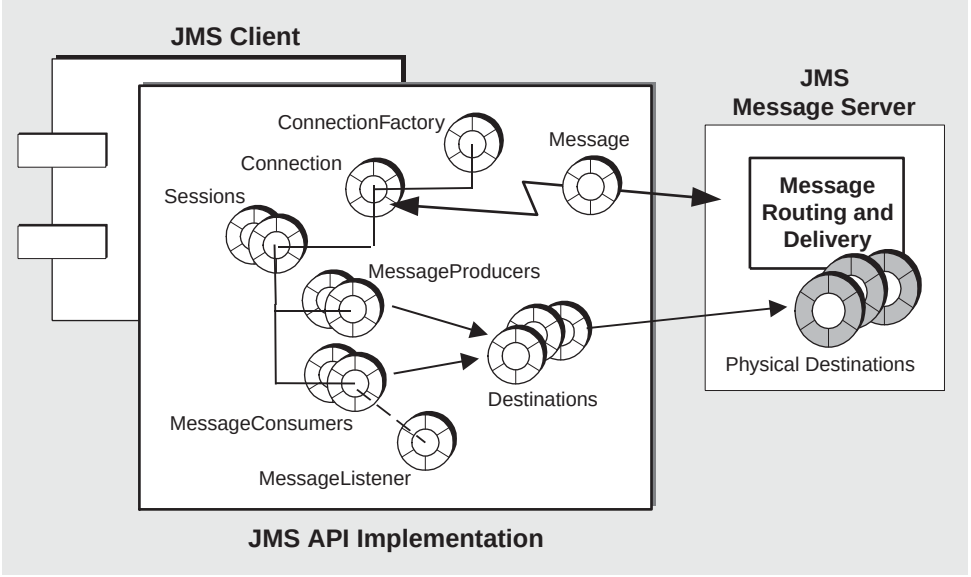
The JMS programming model supports the architecture of an asynchronous messaging service: JMS clients exchange messages by way of a JMS message service. A JMS provider furnishes the objects needed to carry out JMS messaging; these objects implement the JMS application programming interface (API).

This section describes the programming objects needed for JMS messaging and introduces the delivery models (point-to-point and publish/subscribe) used to send and receive messages.

## Programming Objects

The objects used to set up a JMS client for delivery of messages are shown in Figure 1-3.

**Figure 1-3** JMS Programming Objects




In the JMS programming model, a JMS client uses a *connection factory* object (`ConnectionFactory`) to create a *connection* over which messages are sent to and received from a JMS message server. A connection object (`Connection`) represents a client's active connection to the message server.

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.

The connection is used to create *session* objects (`Session`). A session is a single-threaded context for producing and consuming messages. It is used to create messages as well as the message producers and consumers that send and receive them, and it defines a serial order for the messages it delivers. A session supports reliable delivery through a number of *acknowledgement* options or through transactions.

A client uses a message producer object (`MessageProducer`) to send messages to a specified physical *destination*, represented in the API by a destination object. The message producer can specify default message header values, such as delivery mode (persistent vs. non-persistent), priority, and time-to-live, that govern all messages sent by the producer to the physical destination.

Similarly, a client uses a message consumer object (`MessageConsumer`) to receive messages from a specified physical destination, represented in the API as a destination object. There are two types of destination, *queue* and *topic*, depending on the message delivery model.

A message consumer can use a message selector to have the message service deliver only those messages whose properties match specific selection criteria.

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.
- 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 a message when a session thread invokes the `onMessage()` method of the message listener object.

## Programming Domains: Message Delivery Models

JMS supports two distinct message *delivery model*s: point-to-point and publish/subscribe.

**Point-to-point (Queue Destinations)** A message is delivered from a producer to a single consumer. In this delivery model, the destination type is a *queue*. Messages are first delivered to the queue destination, then delivered from the queue, one at a time, to one of the consumers registered for the queue. Any number of producers

can send messages to a queue destination, but each message is guaranteed to be delivered to—and successfully consumed by—only one consumer. If there are no consumers registered for a queue destination, the queue holds messages it receives, and delivers them when a consumer registers for the queue.

**Publish/Subscribe (Topic Destinations)** A message is delivered from a producer to any number of consumers. In this delivery model, the destination type is a *topic*. Messages are first delivered to the topic destination, then delivered to *all* active consumers that have *subscribed* to the topic. Any number of producers can send messages to a topic destination, and each message can be delivered to any number of subscribed consumers.

Topic destinations also support *durable subscriptions.* A durable subscription represents a consumer who is registered with the topic destination but who can be inactive when messages arrive in the destination. The consumer receives the message when it becomes active again. If there are no consumers registered for a topic destination, the topic only holds messages for inactive consumers with durable subscriptions.

These two message delivery models are handled using three sets of API objects—with slightly different semantics—representing different programming *domains*, as shown in Table 1-2.

**Table 1-2** 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 |

* Depending on programming approach, you must specify a particular destination type.

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 offers the advantage of 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.

The example applications included with the Message Queue product as well as many of the code examples in the Message Queue documentation use the separate programming domains.

# Reliable Messaging

The *delivery mode* of a message can be set either to persistent or non-persistent; this mode governs the reliability of message delivery.

- *Persistent messages.* are guaranteed to be delivered and successfully consumed exactly once. Persistent messages are not lost in case of message service failure. Reliability is at a premium for such messages.
- *Non-persistent messages* are guaranteed to be delivered at most once. Non-persistent messages are lost in the case of a message service failure. Reliability is not a major concern for such messages.

There are two aspects of ensuring reliability in the case of persistent messages. One is to ensure, through the use of acknowledgments and transactions, that message production and consumption is successful. The other is to ensure, by placing messages in a persistent store, that the message service does not lose persistent messages before delivering them to consumers.

The following sections describe these two aspects of ensuring reliability.

## Acknowledgements/Transactions

Reliable messaging depends on guaranteeing the successful delivery of persistent messages from a message producer to a physical destination on a message server and from that physical destination to a message consumer. This reliability can be achieved using either of two general mechanisms supported by a JMS session: *acknowledgement*s or *transaction*s. In the case of transactions, these can either be local or distributed (under the control of a distributed transaction manager).

### Acknowledgements

Acknowledgements are messages sent between client and message service to ensure reliable delivery.

In the case of message production, the message service acknowledges that it has received delivery of a message, placed it in its destination and stored it persistently. The producer's `send()` method blocks until the acknowledgement returns.

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 message service deletes the message from that destination. JMS specifies different acknowledgement modes that represent different degrees of reliability. In some of these modes, the client blocks waiting for the message server to confirm that it has deleted a message and therefore cannot redeliver the message.

### Local Transactions

A session can be configured as *transacted*, in which case the production and/or consumption of one or more messages can be grouped into an atomic unit—a *transaction*. 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 local 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 local 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. (In other words, the delivery of a message to a destination and the subsequent delivery of the message to a client cannot be placed in the same transaction.)

### Distributed Transactions

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. 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.

## Persistent Storage

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

A message server must also persistently store durable subscriptions. Otherwise the message server, in case of failure, 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.

# JMS Administered Objects

Two of the objects used in the JMS programming model, connection factories and destinations, can vary with a provider's implementation of the JMS specification.

- The *connection factory object* is used to create connections whose behavior depends on the protocols and mechanisms used by the provider to deliver messages.
- The *destination object* is used to specify the name of physical destinations on the broker and depends on the specific naming conventions and capabilities of the physical destinations on the message server.

To allow providers maximum flexibility in defining these objects while allowing clients to be portable, the JMS specification defines *administered objects* (for connection factories and destinations) that encapsulate provider-specific information. These objects are created and configured by an administrator, stored in a JNDI namespace (object store), and accessed by clients through standard JNDI lookup code.

Administered objects allow JMS clients to use logical names to look up and reference provider-specific objects. In this way, client code does not need to know specific naming or addressing syntax or the configurable properties used by a provider. This makes the code provider-independent.

The section "Administered Objects" on page 42 provides additional information about the administered objects used in Message Queue.

**NOTE** The JMS specification does not require that you access administered objects using a JNDI lookup. Client code can instantiate connection factory and destination objects, and set values for their attributes. However, this means that client code is not portable to other providers.

Chapter 2

# Introduction to Message Queue

Message Queue is a reliable asynchronous messaging service that conforms to the JMS 1.1 specification. In addition, to provide for the needs of large-scale enterprise deployments, Message Queue provides a host of features that exceed JMS specification requirements.

This chapter describes the Message Queue service architecture and introduces its enterprise features and capabilities. The chapter covers the following topics:

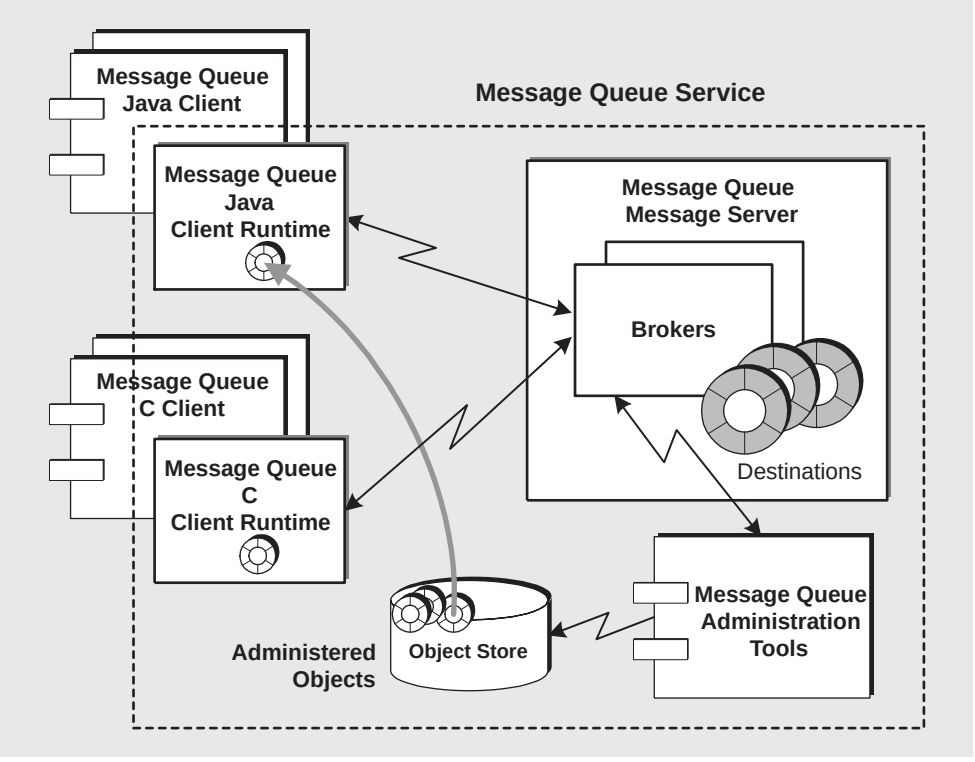
# Message Service Architecture

The Message Queue service is composed of the following elements:

- "Message Server" on page 37
- "Client Runtime" on page 37
- "Administered Objects" on page 42
- "Administration Tools" on page 45

Figure 2-1 shows how these elements work together.

**Figure 2-1** Message Queue Service Architecture




As shown in the figure, a Message Queue client uses the Java or C API to send or receive a message. These APIs are implemented in a Java or C-client runtime library, which does the actual work of creating connections to the broker and packaging the bits appropriately for the connection service requested. If the

application uses administered objects, the client runtime locates these objects in an object store and uses them to configure the connection and to locate physical destinations. The broker routes and delivers the message. An administrator uses Message Queue administrative tools to manage the broker and to add administered objects to the object store.

Each of these elements is described briefly in the following sections.

## Message Server

The message server is composed of one or more brokers and performs message routing and delivery. It is the heart of the Message Queue service.

The message server consists of a single *broker* or a set of brokers working together (as a broker *cluster*) to perform message routing and delivery services. The broker is a process that performs the following tasks:

- The *authentication* of users and *authorization* of the operations they want to perform
- Sets up communication channels with clients
- Receives messages from producing clients and places them in their respective physical destinations
- Routes and delivers messages to one or more consuming clients
- Guarantees reliable delivery
- Provides data for monitoring system performance.

For a detailed description of the message server, its internal components, and the functions they perform, see Chapter 4, “Message Server” on page 71.

Message Queue Enterprise Edition supports the use of broker clusters, consisting of multiple interconnected broker instances, allowing a message server to scale with the volume of message traffic. For a description of architecture and cluster configuration issues, see Chapter 5, “Broker Clusters.”

## Client Runtime

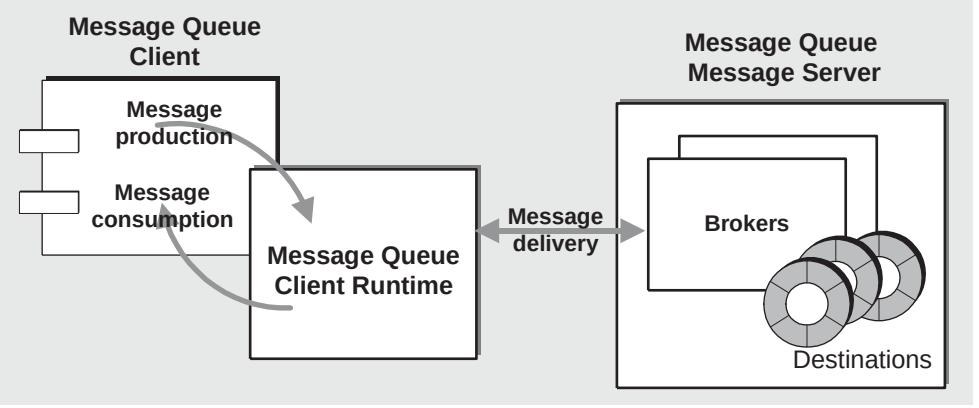
The Message Queue *client runtime* provides client applications with an interface to the Message Queue service. The client runtime supports all operations needed for Message Queue clients to produce messages (send them to destinations) and to consume messages (retrieve them from destinations).

There are two language implementations of the Message Queue client runtime, as shown in Figure 2-1 on page 36:

- **Java client runtime.** Supplies Java client applications and components with all the objects needed to implement the JMS API and to interact with the Message Queue message server. These interface objects include connections, sessions, messages, message producers, and message consumers.
- **C client runtime.** Supplies C client applications and components with the C programming interfaces needed to interact with the Message Queue server. (The C client runtime supports a procedural version of the JMS API messaging model.)

Figure 2-2 illustrates the central role played by the client runtime between Message Queue clients and the message server. Message *production* and *consumption* involve an interaction between clients and the client runtime, while message *delivery* is an interaction between the client runtime and the message server.

**Figure 2-2** Client Runtime and Messaging Operations




The client runtime performs the following functions:

- Manages message delivery to the message server.
- Sets up the connections,
- Establishes the client's identity
- Implements client acknowledgements
- Controls the flow of messages across the connection
- Can override message header values set by producing clients.

The following subsections briefly describe client runtime functions. Some aspects of the behavior of the client runtime can be customized by configuring the properties of the connection factory object.

## Connection Handling

To configure connection handling behavior you must specify the host name and port of the broker to which the client wants to connect and the type of connection service desired. If the connection is made to a broker that is part of a cluster, you must specify a list of addresses to which to make a connection. If one broker is not online, the client runtime can connect you to another broker in the cluster.

In the Enterprise Edition, the client runtime can automatically reconnect to a broker if a connection fails. The reconnection can be to the same broker, or to a broker different from the original connection if the client is connected to a broker that is part of a cluster.

If broker instances do not use a shared, highly available persistent store (as could be achieved through integration of Message Queue with Sun Cluster), persistent messages and other state information held by the failed (or disconnected) broker can be lost if a reconnect is to a different broker instance. That is to say, reconnection provides connection failover but not data availability.

## Client Identification

A client ID can be set on any connection if an application finds it useful; it must be set to identify durable subscribers.

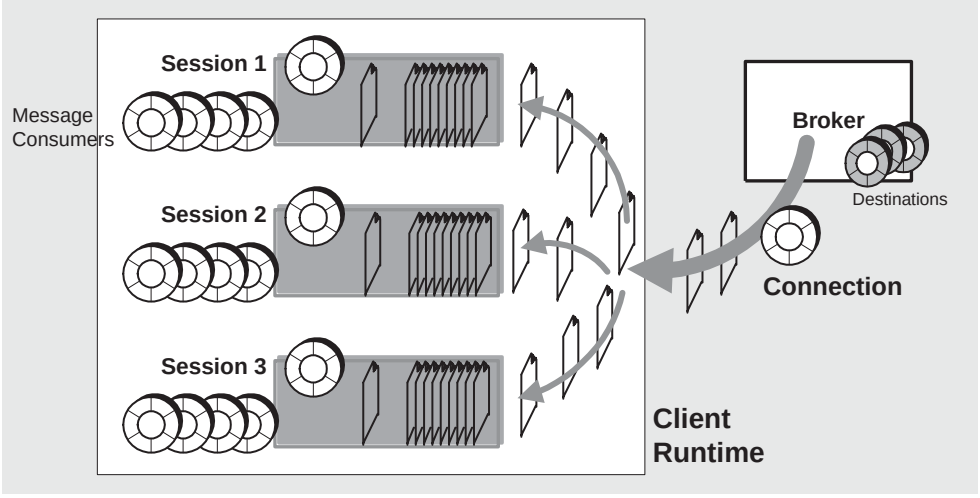
To keep track of durable subscriptions, the broker uses a unique client identification. The client ID is used to identify a durable subscriber that is inactive at the time that messages are delivered to a topic destination. The broker retains messages addressed to such subscribers and makes them available when the subscriber becomes active.

Therefore, a client identifier must be set whenever using durable subscriptions in deployed applications. A Message Queue feature allows you to use a special variable name syntax when you specify the Client ID. This makes it possible to obtain a different client ID for each connection obtained from a connection factory object, whether that object is created by an administrator or programmatically. For more information, see the *Message Queue Administration Guide*.

## Message Distribution to Consumers

Messages delivered by a broker over a connection are received by the client runtime and distributed to the appropriate Message Queue sessions, where they are queued up to be consumed by their respective message consumers, as shown in Figure 2-3 on page 40.

**Figure 2-3** Message Delivery to Message Queue Client Runtime




Messages are fetched off each session queue one at a time and consumed either synchronously (by a client thread invoking the `receive()` method) or asynchronously (by the session thread invoking the `onMessage()` method of a message listener object). (A session is single threaded.)

The flow of messages delivered to the client runtime is metered at a per consumer level. By appropriately adjusting connection factory properties, you can balance the flow of messages so that messages delivered to one session do not adversely affect the delivery of messages to other sessions on the same connection.

## Ensuring Reliable Message Delivery

The client runtime has an important role in ensuring reliable delivery of messages. It supports the client acknowledgement and transaction modes of the JMS specification and controls the various broker acknowledgement behaviors used to guarantee reliable delivery.

The JMS specification describes a number of client acknowledgement modes that provide for different levels of reliability. These acknowledgement modes, and additional modes implemented by Message Queue, are described in the context of message consumption (see "Client Acknowledgements" on page 63).

In the case of persistent messages and reliable delivery, the broker normally acknowledges to the client runtime when it has completed operations used to ensure once and only once consumption of messages. You can use connection factory properties to suppress such broker acknowledgements, thereby saving on network bandwidth and processing. Of course, such suppression of broker acknowledgements, eliminates guarantees of reliable delivery.

## Message Flow Control

The client runtime is the gatekeeper for the flow of messages across a connection. In addition to the regular JMS payload messages that flow across a connection, Message Queue also sends a variety of control messages that are used to guarantee reliable delivery, manage the flow of messages across a connection, and perform other control functions.

Since payload messages and control messages compete for the same connection, they can collide, causing logjams to occur. The client runtime enforces various configurable flow limits and metering schemes to minimize the collision of payload and control messages, and thereby maximize message throughput.

## Overriding Message Header Values

The client runtime can override JMS message header fields that specify the persistence, lifetime, and priority of messages.

Message Queue allows message header overrides at the level of a connection: overrides apply to all messages produced in the context of a given connection.

The ability of the client runtime to override message header values gives a Message Queue administrator more control over the resources of a message server. Overriding these fields, however, has the risk of interfering with application-specific requirements (for example, message persistence). So this capability should be used only in consultation with the appropriate application users or designers.

### Other Functions

The client runtime performs a few other assorted functions:

- **Queue browsing characteristics.** The client runtime can be configured for the number of messages it will retrieve at one time, and the time that it will wait for messages, when browsing the contents of a queue destination.
- **Message compression.** The Java client runtime can compress messages during message production and decompress messages during message consumption. Whether or not such compression or decompression occurs depends on a Message Queue-specific message property set in the message header when the message is created by the client.

## Administered Objects

Administered objects encapsulate provider-specific implementation and configuration information about connections and destinations. Administered objects can be created programmatically, or they can be are created and configured using administrator tools, stored in an object store, and accessed by client applications through standard JNDI lookup code.

Message Queue provides the administered object types shown in the following table.

**Table 2-1** Message Queue Administered Object Types

| Type | Description |
|---|---|
| `Destination` | Represents a physical destination in a broker. Contains the provider-specific name of the physical destination in the broker. Message consumer and/or message producer objects use a destination administered object to access the corresponding physical destination. |
| `Connection Factory` | Establishes physical connections between a client application and a Message Queue message server. Also configures the Message Queue client runtime, which controls the behavior of physical connections. When setting the attribute values of a connection factory administered object, you specify properties that apply to all connections that it establishes. |

**Table 2-1** Message Queue Administered Object Types *(Continued)*

| Type | Description |
|---|---|
| `XA Connection Factory` | Used to establish physical connections that support distributed transactions (see "Distributed Transactions" on page 32). XA connection factory objects share the same set of attributes as regular connection factory objects, but enable the additional mechanisms needed to support distributed transactions. |
| `SOAP Endpoint` | Identifies the final destination of a SOAP message: this is the URL of a servlet that can receive the SOAP message. The SOAP endpoint administered object can be configured to specify multiple URLs. Also specifies the lookup name associated with the object, and object store attributes. |

## Using Administered Objects via JNDI

Although the JMS specification does not require JMS clients to look up administered objects in a JNDI namespace, there are distinct advantages to doing so: it allows for a single source of control, it allows connections (client runtime behavior) to be configured and reconfigured without having to recode, and it allows clients to be portable to other JMS providers.

Administered objects make it easier to control and manage a Message Queue service:

- Administrators can specify the behavior of the client runtime by requiring client applications to access preconfigured connection factory objects.
- Administrators can control the proliferation of physical destinations by requiring client applications to access preconfigured destination administered objects that correspond to existing physical destinations.

In other words, the use of administered objects allows a Message Queue administrator to control message service configuration details, while at the same time allowing client applications to be provider-independent.

Using administered objects means that client programmers do not have to know about provider-specific syntax and object naming conventions or provider-specific configuration properties. In fact, by specifying that administered objects be read only, administrators can ensure that client applications cannot change administered object attribute values that were set when the administered object was first created.

While it is possible for client applications to instantiate both connection factory and destination administered objects on their own, this practice undermines the basic purpose of an administered object. Message Queue administrators need to control broker resources required by an application and to tune messaging performance. In addition, directly instantiating administered objects makes client applications provider-dependent.

Notwithstanding these arguments, applications often instantiate administered objects in development environments in which administrative control is not an issue.

## Object Stores

Message Queue administered objects are placed in an object store (see Figure 2-1 on page 36) where they can be accessed by client applications through a JNDI lookup. Message Queue supports two types of object store: a standard LDAP directory server and a file-system object store.

**LDAP Server Object Store** An LDAP server is the recommended object store for production messaging systems. LDAP implementations are available from a number of vendors and are designed for use in distributed systems. LDAP servers also provide security features that are useful in production environments.

**File-system Object Store** Message Queue supports a file-system object store, which is not recommended for production systems but has the advantage of being very easy to use in development environments. Rather than setting up an LDAP server, all you have to do is create a directory on your local file system. A file-system object store, however, cannot be used as a centralized object store for clients deployed across multiple computer nodes unless these clients have access to the directory where the object store resides.

# Administration Tools

Message Queue administration tools consist of a set of command line utilities and a graphical user interface (GUI) Administration Console.

**Command Line Utilities** Message Queue provides a set of command line utilities to perform all Message Queue administration tasks, such as starting up and managing a broker, creating and managing physical destinations, managing administered objects, and performing other, more specialized administrative tasks. All the command line utilities share common formats, syntax conventions, and options. For more detailed information on the use of the command line utilities, see the *Message Queue Administration Guide*.

**The Administration Console** The Console provides a subset of the capabilities of the Message Queue command line utilities. You can use the Administration Console to manage a broker, create and manage physical destinations, and manage administered objects. However you cannot perform the more specialized tasks of some of the command line utilities. For example, you cannot use the Administration Console to start up a broker, create broker clusters, or manage a user repository. These tasks must be performed using the Message Queue command line utilities.

The *Message Queue Administration Guide* provides a brief, hands-on tutorial to familiarize you with the Administration Console and to illustrate how you use it to accomplish basic tasks.

The Administration Console and some of the command line utilities allow for remote management of brokers and physical destinations.

# Product Features

The Message Queue service, and the architecture described in the previous section, fully implement the JMS 1.1 specification for reliable, asynchronous, flexible message delivery. For documentation of JMS compliance-related issues, see *Appendix A, "Message Queue Implementation of Optional JMS Functionality."*

However, 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.

Message Queue's enterprise features, discussed below, are grouped into the following categories:



## Integration Support Features

Message Queue allows you to integrate disparate applications and components across an enterprise by including support for several transport protocols, a C client interface to the Message Queue service, support for SOAP (XML) messages, and a pluggable J2EE resource adapter.

### Multiple Transport Support

Message Queue supports the ability of clients to interact with the Message Queue message server over a number of different transport protocols, including TCP and HTTP, and using secure connections.

**HTTP connections** 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 delivered over HTTP through a firewall to the tunnel servlet. The tunnel servlet extracts the message from an HTTP request and delivers the message over TCP/IP

to the broker. In a similar fashion, Message Queue supports secure HTTP connections using an HTTPS tunnel servlet. For more information on the architecture of HTTP connections, see "HTTP/HTTPS Support" on page 76. For information on setting up and configuring HTTP/HTTPS connections, see the *Message Queue Administration Guide*.

**Secure connections** Message Queue provides for secure transmission of messages based on 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.

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. For information on creating self-signed certificates to enable SSL-based connections services, see the *Message Queue Administration Guide*.

## C Client Interface

In addition to supporting Java language messaging clients, Message Queue also provides a C language interface to the Message Queue service. The C API enables legacy C applications and C++ applications to participate in JMS-based messaging. However, clients using Message Queue's C API are not portable to other JMS providers.

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.

For more information on the features of the C API and how it implements the JMS programming model with C data types and functions, see the *Message Queue Developer's Guide for C Clients*.

## SOAP (XML) Messaging Support

Message Queue supports creation and delivery of messages that conform to the Simple Object Access Protocol (SOAP) specification. SOAP allows for the exchange of structured XML data, or SOAP messages, between peers in a decentralized, distributed environment. A SOAP message is an XML document that can also contain an attachment, which does not have to be in XML.

The fact that SOAP messages are encoded in XML makes SOAP messages platform independent. They can be used to access data from legacy systems and share data between enterprises. The data integration offered by XML also makes this technology a natural for Web-based computing, such as Web services. Firewalls can recognize SOAP packets and can filter messages based on information exposed in the SOAP message header.

Message Queue implements the SOAP with Attachments API for Java (SAAJ) specification. SAAJ is an application programming interface that can be implemented to support a programming model for SOAP messaging and to furnish Java objects that you can use to construct, send, receive, and examine SOAP messages. SAAJ defines two packages:

- `javax.xml.soap`: You use the objects in this package to define the parts of a SOAP message and to assemble and disassemble SOAP messages. You can also use this package to send a SOAP message without the support of a provider.
- `javax.xml.messaging`: You use the objects in this package to send a SOAP message using a provider and to receive SOAP messages.

Message Queue provides utilities to transform SOAP messages into JMS messages and *vice versa*. These utilities allow SOAP messages to be received by a servlet, transformed into a JMS message, delivered by the Message Queue service to a JMS consumer, transformed back into a SOAP message, and delivered to a SOAP endpoint. In other words, Message Queue supports the ability to reliably and asynchronously exchange SOAP messages between SOAP endpoints or, more simply, to publish SOAP messages to Message Queue subscribers.

For additional information, see the *Message Queue Developer's Guide for Java Clients*.

## J2EE Resource Adapter

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. In short, the J2EE platform requires JMS support.

This support is provided in the J2EE programming model using the message-driven bean (MDB), a specialized type of Enterprise Java Bean (EJB) component that can consume JMS messages. A J2EE-compliant application server must provide an MDB container that supports JMS messaging. This can be achieved by plugging-in a JMS resource adapter into the application server. Message Queue provides such a resource adapter.

By plugging the Message Queue resource adapter into an application server, 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 6, "Message Queue and J2EE."

## Security Features

Protecting stored and in-transit message data is critical for most enterprise applications. Message Queue provides security at many levels, including authentication of users, controlled access to resources, and message encryption.

**Authentication** Message Queue supports password-based authentication of users. Connections to the message server are granted to users based on passwords stored in a flat file or LDAP user repository. Information about all connection attempts (users and host computers) is logged and can be tracked.

**Authorization** Access control lists (ACLs) provide configurable, fine-grained control over access to a broker's connections and physical destinations. Both user and group access is supported. Authorization is performed on a broker-by-broker basis; each broker can have a different access control file.

**Encryption** SSL support allows all message traffic between a message server and its clients (whether over TCP/IP or HTTP connections) to be encrypted using a full-strength SSL implementation.

For information on populating a user repository, managing access control lists, and setting up SSL support, see the *Message Queue Administration Guide*.

# Scalability Features

Message Queue allows you to scale your application as users, client connections, and message loads grow.

## Scalable Connection Capacity

The Message Queue broker can handle thousands of concurrent connections. By default, each connection is handled by a dedicated broker thread. Because this ties up the thread even when the connection is idle, you can configure the connection service so that multiple connections can share the same thread. This shared threadpool model can dramatically expand the number of connections that a broker can support. For more information, see "Thread Pool Manager" on page 75.

## Broker Clusters

As the number of connections and the number of messages being delivered through a broker increases, the extra load can be managed by adding additional broker instances to the Message Queue server. Broker clusters balance client connections and message delivery across a number of broker instances, making the message server highly scalable. The broker instances can be on the same host or distributed across a network. Clustering is an ideal way to improve message throughput and expand messaging bandwidth as business needs grow. Broker clusters are introduced in Chapter 5, "Broker Clusters" on page 91 and discussed more fully in *Message Queue Administration Guide*.

## Queue Delivery to Multiple Consumers

According to the JMS specification, a message in a queue destination can be delivered only to a single consumer. Message Queue allows multiple consumers to register with a queue. The broker can then distribute messages to the different registered consumers, balancing the load among them and allowing the system to scale.

The implementation of queue delivery to multiple consumers uses a configurable load-balancing approach. Using this approach, you can specify the maximum number of active consumers and the maximum number of backup consumers standing by to take the place of active consumers should any fail. In addition, the load-balancing mechanism takes into account a consumer's current capacity and message processing rate.

For more information on load-balanced queue delivery, see "Queue Delivery to Multiple Consumers" on page 61.

# Availability Features

Message Queue provides a number of features for minimizing service downtime. These range from mechanisms for preventing failure to those that allow integration with Sun Cluster to provide high availability.

## Message Service Stability

One of the most effective ways of ensuring availability of a message service is to provide a service that offers high performance and minimizes failure. Message Queue provides mechanisms for averting memory overloads or performance logjams. These operate on both the message server and client runtime.

**Message server resource management** Because the message server is limited in memory and CPU resources, it is possible for it to become overloaded to the point where it becomes unresponsive or unstable. This commonly happens when the rate of message production far exceeds the rate of consumption. To avert such situations, a broker can be configured on the level of individual physical destinations and on a system-wide level to prevent memory overruns. For more information, see "Memory Resource Management" on page 79.

**Client runtime message flow control** In addition, Message Queue provides mechanisms for controlling the delivery of messages to the client runtime. You can use flow control mechanisms to optimize the delivery of messages to the client runtime while preventing the client from running out of memory. For more information, see "Message Flow Control" on page 41.

## Automatic Reconnect to Message Server

Message Queue provides an automatic reconnect capability: If a connection between a message server and client fails, Message Queue maintains the client state while attempting to reestablish the connection. In most cases, message production and consumption will transparently resume once the connection is re-established. for more information, see *Message Queue Administration Guide*.

## High Availability Through Sun Cluster

While Message Queue's broker clustering provides a highly scalable message server, it does not currently support failover from one broker instance in a cluster to another. However, Message Queue can be integrated with Sun Cluster software to provide a high-availability message server. Using a Sun Cluster agent developed for Message Queue, Sun Cluster can ensure that no state data is lost if a broker fails, allowing a message server to be restored immediately and transparently with virtually no downtime.

# Manageability Features

Message Queue provides a number of features that you can use to monitor and administer a message service and to tune message service performance.

## Robust Administration Tools

Message Queue offers both command line and GUI tools for administering a Message Queue message server and for managing destinations, transactions, durable subscriptions, and security (see "Administration Tools" on page 45).

Message Queue also supports remote monitoring and administration of message servers as well as tools for managing JMS administered objects, user repositories, plugged-in JDBC-compliant data stores, and self-signed server certificates. For information on using these administration tools, see the *Message Queue Administration Guide*.

## Message-Based Monitoring API

Message Queue provides a simple JMS-based monitoring API that you can use to create custom monitoring applications. These monitoring applications are consumers that retrieve metrics messages from special topic destinations. The metrics messages contain monitoring data provided by the Message Queue broker (see "Metrics Message Producer (Enterprise Edition)" on page 86).

For details of the metrics quantities reported in each type of metrics message, see the *Message Queue Developer's Guide for Java Clients*, which explains how to develop a Message Queue client for consuming metrics messages. For information about how to configure the production of metrics messages, see the *Message Queue Administration Guide*.

## Tunable Performance

Message Queue offers many ways to tune both the message server and the client runtime to achieve optimal performance. You can monitor key resources and adjust memory usage, threading resources, message flow, connection services, reliability parameters, and other elements that affect message throughput and system performance. For details about how to tune message service performance, see the *Message Queue Administration Guide*.

# Flexible Server Configuration Features

Message Queue allows you to choose how persistent objects, user information, and administered objects are stored.

## Configurable Persistence

In order to guarantee delivery of messages, Message Queue stores messages and other persistent objects until messages are consumed. In addition to providing a high performance file-based persistent store, Message Queue also supports configurable persistence. This allows you to store persistent messages in embedded or external JDBC-compliant databases, such as Oracle 8i. For more information, see "Persistence Manager" on page 81.

## LDAP Server Support

Message Queue provides file-based storage for both administered objects and user information needed for authentication and authorization. However, Message Queue also supports using LDAP servers for administered object stores and user repositories. LDAP servers provide a more secure, standard way of storing and retrieving such information, and are recommended for production systems. For information on using LDAP servers for administered object stores and user repositories, see the *Message Queue Administration Guide*.

# Product Editions

Message Queue is available in two editions: Enterprise and Platform. Both editions provide a full implementation of the JMS specification, but each corresponds to a different feature set and licensed capacity. The feature sets are compared in the following table. For a description of the features, see "Product Features" on page 46.

**Table 2-2** Feature Comparison: Enterprise and Platform Editions

| Enterprise Edition | Platform Edition |
|---|---|
| *Advanced Integration Support Features* | |
| HTTP support, TCP support | TCP support |
| Secure connections based on Secure Socket Layer (SSL) standard | Secure connections based on Secure Socket Layer (SSL) standard |
| C language API, Java API | Java API<br>(The C API can only be used with the trial license.) |
| SOAP (XML) messaging support | SOAP (XML) messaging support |
| J2EE resource adapter | J2EE resource adapter |
| *Security Features* | |
| Authentication from either a flat file or LDAP user repository, authorization using an access control file, and SSL encryption. | Same as Enterprise Edition |
| *Scalability Features* | |
| Scalable connection capacity | Fixed connection capacity |
| Message server can be implemented as a broker cluster | Single-broker message server |
| Queue delivery to unlimited number of message consumers (per queue) | Queue delivery to a maximum of three message consumers (per queue) |

**Table 2-2** Feature Comparison: Enterprise and Platform Editions *(Continued)*

| Enterprise Edition | Platform Edition |
|---|---|
| *Availability Features* | |
| Message service stability through memory resource management and message flow control | Same as Enterprise Edition |
| Client connection failover to a different broker in a cluster or automatic reconnect to the same broker. | No client connection failover. Automatic reconnect to same broker allowed. |
| High availability through Sun Cluster | High availability through Sun Cluster |
| *Manageability Features* | |
| Robust administration tools | Robust administration tools |
| Message-based monitoring API in addition to administration tools and logging | Administration tools and logging, but no message-based monitoring API |
| Tunable performance | Tunable performance |
| *Flexible Server Configuration Features* | |
| Pluggable Persistence | Pluggable Persistence |
| LDAP server support | LDAP server support |

The license capacities of the Platform and Enterprise Editions are described below.

# Enterprise Edition

The Message Queue enterprise edition allows you to deploy and run messaging applications in an enterprise production environment. You can also use it for developing, debugging, and load-testing messaging applications and components. The Enterprise Edition has an unlimited duration license based on the number of CPUs that are used. The license places no limit on the number of brokers in a multi-broker message service.

# Platform Edition

The Message Queue Platform Edition places no limit on the number of client connections supported by the message server. It comes with a basic license or a 90-day trial license:

- **A basic license** has an unlimited duration. Platform Edition with a basic license can be used as a JMS provider in less demanding production environments. This license does *not* include Enterprise Edition features.
- **A 90-day trial enterprise license** includes all Enterprise Edition features not included in the basic license. However, the license has a limited 90-day duration enforced by the software, making it suitable for evaluating the features available in the Enterprise Edition. For instructions on using the 90-day trial enterprise license, see the startup options discussed in the *Message Queue Administration Guide*.

The Platform Edition can be downloaded free from the Sun web site and is also bundled with the Sun Java System Application Server platform. Instructions for upgrading Message Queue from Platform Edition to Enterprise Edition can be found in the *Message Queue Installation Guide*.

**NOTE** For all editions of Message Queue, a portion of the product—the Message Queue client runtime—can be freely redistributed for commercial use. All other files in the product *cannot* be redistributed. The portion that can be freely redistributed allows a licensee to develop a Message Queue client application (one which can be connected to a Message Queue service) that can be sold to a third party without incurring any licensing fees. The third party will either need to purchase Message Queue to access a Message Queue message server or make a connection to yet another party that has a Message Queue message server installed and running.

# Message Queue in a Sun Product Context

Besides being the middleware directly used by applications, Message Queue is also used by other middleware as well as by other servers and applications delivered by Sun. To facilitate this, Message Queue is delivered in Solaris and Java Enterprise System as well as being delivered in the Sun Java System Application Server.

In the Application Server, Message Queue satisfies the JMS requirement that the J2EE platform furnish a JMS provider. It is used directly by the applications that are hosted by the Application Server. For more information, see Chapter 6, “Message Queue and J2EE” on page 97.

Chapter 3

# Reliable Message Delivery

This chapter describes how the Message Queue service provides reliable message delivery. It traces the path of a message through the system, describing the various mechanisms used to route and deliver the message to the appropriate consumer, and to guarantee that it has been delivered.

The chapter covers the following topics:

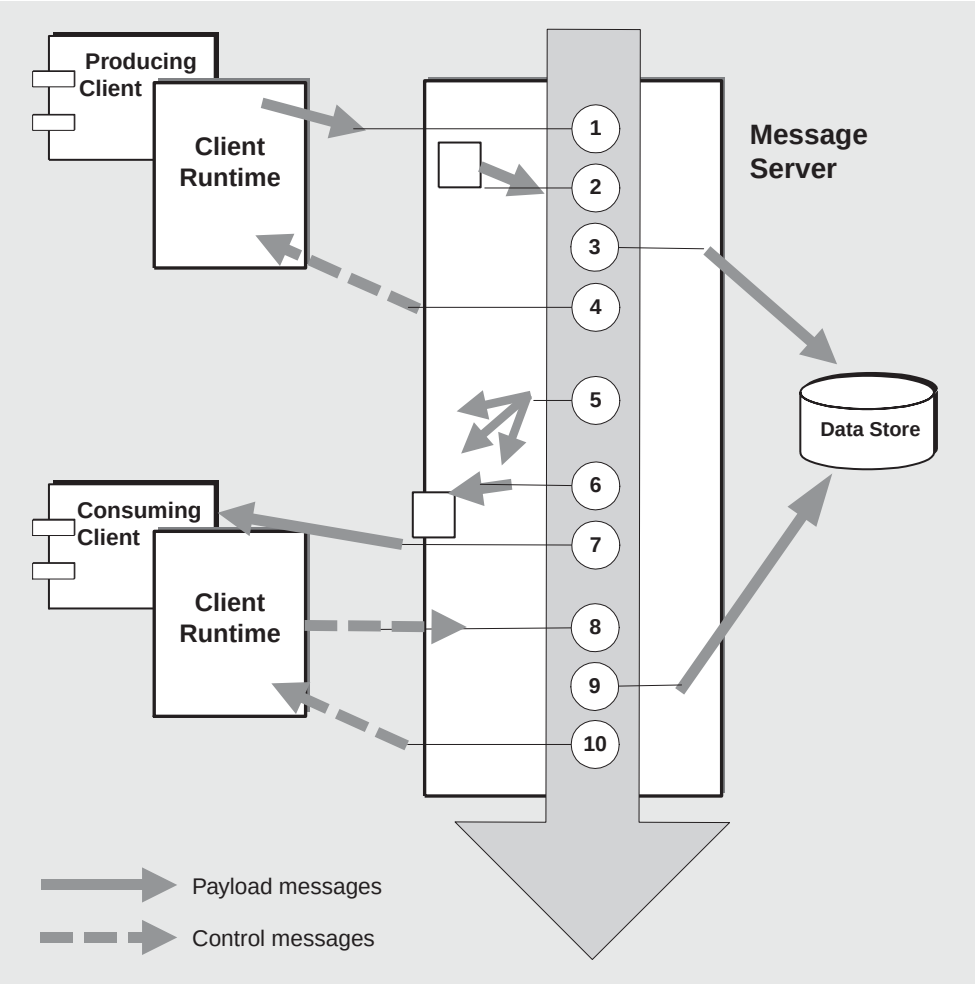


This chapter has material of interest to both developers and administrators and supplements the information in Chapter 2, "Introduction to Message Queue."

# A Message's Journey Through the System

The delivery of a message by the Message Queue message service, from a message producer to a message consumer is illustrated in Figure 3-1. The subsections that follow provide a more detailed description of each stage in the delivery process.

**Figure 3-1** 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 message server.

#### Message Handling and Routing

**2.** The message server reads in the message from the connection and places it in the appropriate destination.

**3.** The message server places the (persistent) message in the data store.

**4.** The message server acknowledges receipt of the message to the client runtime of the message producer.

**5.** The message server determines the routing for the message.

**6.** The message server writes out the message from its destination to the appropriate connection.

#### 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 message server.

#### Message End-of-Life

**9.** The message server processes the client acknowledgement, deleting the (persistent) message from both its destination and the data store.

**10.** The message server confirms to the consumer's client runtime that the client acknowledgement has been processed and the message cannot be delivered again.

The messages handled by the system in the course of these delivery steps fall into two categories:

- **Payload messages**. The JMS messages sent by producing clients to consuming clients.
- **Control messages**. Acknowledgements and other non-payload messages passed between message server and client runtime to guarantee that payload messages are successfully delivered and to control the flow of messages across a connection.

# Message Delivery Processing

The processing of a message by a Message Queue service, in the course of its delivery from producer to consumer, proceeds in several stages, as shown in the description of the steps following Figure 3-1.

The stages are as follows:

- "Message Production" on page 60
- "Message Handling and Routing" on page 61
- "Message Consumption" on page 63
- "Message-End-of-Life" on page 66

These stages are described in the following sections.

## Message Production

In message production, a message is created by the client and sent by the client runtime over a connection to a destination on a broker.

If the message's delivery mode has been set to persistent (guaranteed delivery, once and only once, even if the broker fails), the broker, by default, sends a control message—a broker acknowledgement—back to the client runtime. This broker acknowledgement indicates that the broker delivered the message to its destination and stored it in the broker's data store. The client thread blocks until it receives the broker acknowledgement.

If the message's delivery mode has been set to non-persistent, the broker, by default, does not send a broker acknowledgement back to the client runtime and the client thread does not block. However, if it is important to know whether the broker receives non-persistent messages, you can enable broker acknowledgement. In fact, broker acknowledgement must be enabled for the broker to slow message production when destination memory limits are reached (see "Destination Message Limits" on page 80).

# Message Handling and Routing

When the broker receives an incoming JMS payload message, it places it in its target destination and then routes it to the appropriate consumer or consumers.

In general, all messages remain at their physical destination (in memory) until they are delivered or expire. If the broker should fail, these messages would be lost. If a message is persistent, the broker stores it in a database or file system and recovers it after a failure.

The handling of a message depends on its destination type—queue or topic—as described in the following sections. It also depends on destination properties that are set for the destination when the administrator creates the physical destination.

## Queue Destinations

Queue destinations are used in point-to-point messaging, where a message is meant for delivery to and consumption by only one consumer.

While any message in a queue is delivered to only a single consumer, Message Queue allows multiple consumers to register with a queue. The broker can then distribute messages to the different registered consumers, balancing the load among them.

### *Basic Routing Mechanisms*

Messages are queued as they arrive from producers. As each message reaches the front of the queue, it is routed to a single consumer registered with the queue. The order in which a message reaches the front of the queue depends on the order of its arrival and on its priority.

If a selector property value has been set in a message, the broker compares it to any selector values specified by the registered consumer, and ensures that the selector values match before routing the message to the consumer.

### *Queue Delivery to Multiple Consumers*

The implementation of queue delivery to multiple consumers uses a configurable load-balancing approach based on a number of queue destination properties:

- You can set the maximum number of consumers that are active in load-balanced queue delivery.
- You can set the maximum number of backup consumers that can take the place of active consumers should any of them fail.

New consumers are rejected if the number of consumers exceeds the sum of these two properties. (Message Queue Platform Edition supports up to three consumers per queue—two active and one backup—and Message Queue Enterprise Edition supports an unlimited number.)

The load-balancing mechanism takes into account the message consumption rate of different consumers. Messages in a queue destination are routed to newly available active consumers (in the order in which they registered with the queue) in batches of a configurable size (the queue destination's consumer flow limit property). Once these messages have been delivered, additional messages arriving in the queue are routed in batches to consumers as consumers become available. A consumer becomes available when it has consumed a configurable percentage of messages previously delivered to it. In other words, the dispatch rate to each consumer depends on the consumer's current capacity and message processing rate.

If an active consumer fails, then the first backup consumer is made active and takes over the work of the failed consumer. Because of these mechanisms, if a queue destination has more than one active consumer, no guarantee can be made about the order in which messages are consumed.

When the rate of message production is slow, the broker might dispatch messages unevenly among active consumers. If you have more active consumers than necessary, some may never receive messages.

In a broker cluster environment, you can set delivery to multiple consumers to prioritize local consumers. You can use a queue destination property to specify that messages be delivered to remote consumers only if there are no consumers on a producer's home broker—that is, the broker to which the producer sent its messages (the local broker). This lets you increase performance in situations where routing to remote consumers (through *their* home brokers) might cause slowdowns in throughput.

## Topic Destinations

Topic destinations are used in publish/subscribe messaging, where a message is meant for delivery to all consumers that have registered an interest in the destination.

### *Basic Routing Mechanisms*

As a message arrives from a producer, it is routed to all consumers subscribed to the topic. If consumers have registered a durable subscription to the topic, they do not have to be active when the message arrives to receive the message: the broker will store the message until the consumer is once again active, and then deliver the message.

If a selector property value has been set in a message, the broker compares it to any selector values specified by the registered consumer, and ensures that the selector values match before routing the message to the consumer.

### *Durable Subscriptions and Client Identifiers*

Only one user may have a durable subscription to a topic. As that user opens and closes connections to the message server, the user's identity must remain the same. A *client identifier* is used to make sure that each durable subscription corresponds to only one user.

A client identifier associates a client's connection to a message server with state information maintained by the message server on behalf of the client. By definition, a client identifier is unique.

To create a durable subscription, a client identifier must be either programmatically set by the client, using a JMS API method call, or administratively configured in the connection factory objects used by the client.

## Message Consumption

Once messages have been routed, they are delivered to their respective consumers. When a consumer receives a payload message, the consuming client runtime sends the broker an acknowledgement that the message has been received and processed by the client. The broker waits for this client acknowledgement before deleting the message from its destination. Client acknowledgements can apply to individual messages, groups of messages, or transactions.

### Client Acknowledgements

In accordance with the JMS specification, a client can specify one of three basic acknowledgement modes when creating a session. Which mode you choose depends on the message delivery reliability desired:

Message Queue extends the set of client acknowledgement modes with the addition of a `NO_ACKNOWLDEGE` mode. The basic and extended modes are described in the following subsections.

### *AUTO_ACKNOWLEDGE Mode*

In the AUTO_ACKNOWLEDGE mode, the session automatically acknowledges each message consumed by the client. In addition, the session thread blocks, waiting for the broker to confirm that it has processed the client acknowledgement for each consumed message. This confirmation, in turn, is called a broker acknowledgement.

- For asynchronous message consumption by a message listener, the message is acknowledged after the onmessage() method returns.
- For synchronous message consumption, the message is acknowledged just before the receive() method returns. In this case, it is possible for a message to be partially processed but lost, if the system fails before the message is consumed. For increased reliability, use CLIENT_ACKNOWLEDGE mode or a transacted session to guarantee that no message is lost if the system fails.

### *CLIENT_ACKNOWLEDGE Mode*

CLIENT_ACKNOWLEDGE mode gives the client the most control. In this mode, the client explicitly acknowledges after one or more messages have been consumed. The acknowledgement takes place when the client calls the acknowledge() method of a message object, causing the session to acknowledge all messages that have been consumed by the session since the previous invocation of the method. (This could include messages consumed asynchronously by many different message listeners in the session, independent of the *order* in which they were consumed.)

In addition, the session thread blocks, waiting for the return broker acknowledgement for the batch of consumed messages, which confirms that the broker has processed the client acknowledgement.

Because client acknowledgements and broker acknowledgements are generally batched (rather than being sent one by one), CLIENT_ACKNOWLEDGE mode generally conserves connection bandwidth and reduces the overhead for broker acknowledgements, as compared with AUTO_ACKNOWLEDGE mode. Of course, if in this mode, the client acknowledges each message, no batching will occur, and the acknowledgements are sent one by one.

---

**NOTE** Message Queue also provides a specific method you can use in CLIENT_ACKNOWLEDGE mode, by which you can acknowledge only the *individual* message on which you invoke the method, rather than the standard behavior. This is achieved using programming techniques described in the *Message Queue Developer's Guide for Java Clients*.

---

### *DUPS_OK_ACKNOWLEDGE Mode*

In DUPS_OK_ACKNOWLEDGE mode, the session acknowledges after ten messages have been consumed. This value is not currently configurable. Unlike AUTO_ACKNOWLEDGE or CLIENT_ACKNOWLEDGE modes, the session thread does not block waiting for a broker acknowledgement, because no broker acknowledgement is requested in the DUPS_OK_ACKNOWLEDGE mode.

This means there is no guarantee that messages are delivered and consumed only once. In general, messages will not be redelivered very often; they are redelivered only in cases of failure, where the broker has not received a client acknowledgement for a message it has delivered. Clients should use DUPS_OK_ACKNOWLEDGE mode if they don't care about duplicate delivery.

Because client acknowledgements are batched and the client thread does not block, message throughput is generally much higher than in other modes.

### *NO_ACKNOWLEDGE Mode*

In NO_ACKNOWLEDGE mode, the broker performs the client acknowledgement on behalf of the client, so there is no guarantee that a message has been successfully processed by the consuming client.

Use this mode when message throughput is important but reliable delivery is not. This might be the case, for example, when messages are sent periodically at short intervals, so message load is high and lost messages do not matter a lot.

This mode extends the JMS specification and should only be used by clients that do not need to work with other JMS providers.

## Transactions

The client and broker acknowledgement processes described above apply, as well, to JMS message deliveries grouped into transactions. In such cases, client and broker acknowledgements operate on the level of the transaction, including all messages involved in the transaction. When a transaction commits, a broker acknowledgement is sent automatically.

The broker tracks transactions, allowing them to be committed or rolled back should they fail. This transaction management also supports local transactions that are part of larger, distributed transactions (see "Distributed Transactions" on page 32). The broker tracks the state of these transactions until they are committed. When a broker starts up, it inspects all uncommitted transactions and, by default, rolls back all transactions except those in a PREPARED state, which must be resolved manually.

Message Queue implements support for distributed transactions through an XA connection factory, which lets you create XA connections, which in turn let you create XA sessions. In addition, support for distributed transactions requires either a third party Java Transaction Service (JTS) or a J2EE-compliant Application Server (that provides JTS).

# Message-End-of-Life

The broker deletes messages from destination memory after they are successfully delivered. However, there are times when a message might be discarded without having been successfully delivered. The following subsections describe the conditions under which messages are discarded.

## Normal Deletion of Messages

Under normal conditions, the broker deletes a message from destination memory when the message has been successfully delivered, as confirmed by a client acknowledgement.

When a broker delivers a message to a consumer, it marks the message as delivered, but does not really know whether it has been received and consumed. Therefore, the broker waits for a client acknowledgment before deleting the message from its physical destination and from persistent store.

If the message is sent to a topic, the broker does not delete it until it has received a client acknowledgement from each message consumer to which it has delivered the message. In the case of durable subscriptions to a topic, the broker retains each message in that destination, delivering it as each durable subscriber becomes an active consumer. The broker records client acknowledgements as it receives them and deletes the message only after all the acknowledgements have been received (unless the message expires before then). Depending on the client acknowledgement mode, the broker may confirm receipt of the client acknowledgement by sending a broker acknowledgement back to the client.

If a broker or the connection were to fail, the broker might not receive a client acknowledgement and will redeliver all previously delivered but unacknowledged messages, marking them with a `Redelivered` flag. For example, if a queue consumer goes off line before acknowledging receipt of a message, and another consumer (or even the same consumer) subsequently registers with the queue, the broker will redeliver the unacknowledged message to the new consumer, marking it with a `Redelivered` flag. Client applications concerned about redelivery of a message under such conditions should check messages for the this flag.

**NOTE** There is a JMS API (`recover Session`) by which a client can explicitly request redelivery of messages that have been received but not yet acknowledged by the client. When redelivering such messages, the broker marks them with a `Redelivered` flag.

## Abnormal Deletion of Messages

When a message cannot be delivered, it is either discarded or placed on the dead message queue, depending on the conditions that prevented its delivery.

A message would be discarded by the broker before having been successfully delivered and consumed under the following conditions:

- You use administration tools to purge a destination of one or more messages
- You remove or redefine a durable subscription, leaving a message in a topic destination without the possibility of delivery.

However, under these conditions, a message would be considered dead and either discarded or placed on the dead message queue, depending on the behavior you configure:

- A message expires, exceeding the value of `JMSExpiration` set in its message header.
- A destination's "remove" limit behavior is invoked because the destination exceeded a memory limit threshold.
- Message delivery fails because a message cannot be consumed (an exception is thrown by the client).

You can choose to retain such messages and have them placed in a dead message queue. When placing messages in the dead message queue, the broker writes Message Queue-specific property values to the message, specifying the time and reason for placing them there.

You can subsequently retrieve messages from the dead message queue for diagnostic purposes. See "The Dead Message Queue." on page 79 for more information.

# Performance Issues

The more reliable the delivery of messages, the more overhead and bandwidth are required to achieve it. The trade-off between reliability and performance is therefore a significant design consideration. You can maximize *performance* by choosing to produce and consume non-persistent messages. On the other hand, you can maximize reliability by producing and consuming persistent messages and using transacted sessions. Between these extremes are a number of options, depending on the needs of each application.

The rate at which messages can be processed, for example, is a product of many factors, including messaging application design, configuration of the message server, and configuration of the client runtime. Although these factors are quite distinct, their interactions can complicate the task of maximizing performance.

This section briefly reviews a few of the factors that figure into the trade-off between reliability and performance.

**Delivery Mode** The delivery mode specifies whether a message is to be delivered at most once (non-persistent) or once and only once (persistent). The management of persistent messages requires the use of broker acknowledgement messages flowing across a connection and the use of client acknowledgement modes that block, waiting to receive the broker acknowledgements. To increase throughput, you can set the client runtime to suppress broker acknowledgements, but this eliminates the guarantee that persistent messages are delivered once and only once.

**Client Acknowledgement Mode** Each of the four client acknowledgement modes requires a different level of processing and bandwidth overhead. `AUTO_ACKNOWLEDGE` mode consumes the most overhead and guarantees reliability on a message-by-message basis, `CLIENT_ACKNOWLEDGE` mode batches acknowledgements and therefore requires less bandwidth overhead, while `DUPS_OK_ACKNOWLEDGE` mode consumes the least overhead but allows for duplicate delivery of messages. `NO_ACKNOWLEDGE` mode gives the best performance at the cost of possible message loss.

**Client Application Design** The number of messages queued up in a session is a function of the number of message consumers using the session and the message load for each consumer. If a client is slow in producing or consuming messages, you can normally improve performance by redesigning the application to distribute message producers and consumers among a greater number of sessions or to distribute sessions among a greater number of connections. Design issues that affect performance are discussed in the *Message Queue Developer's Guide for Java Clients* and the *Message Queue Developer's Guide for C Clients*.

**Message Flow Metering** The contention between control messages and payload messages for connection bandwidth can be managed by the client runtime. Configuring the client runtime appropriately can help speed the delivery of broker acknowledgements, freeing blocked session threads and speeding up consumption of messages. See the *Message Queue Administration Guide* for more information.

**Message Flow Limits** Message consumption can be slowed when client runtime resource limitations are approached. By limiting the number of messages held in the client runtime waiting to be consumed by one or more consumers, these resource limitations can be avoided. See the *Message Queue Administration Guide* for more information.

Chapter 4

# Message Server

The Message Queue message server, introduced in "Message Server" on page 37, consists of a single *broker* or a set of brokers working in concert (a broker cluster) to perform message routing and delivery.

This chapter discusses the internal structure of the broker, describes its various components, and outlines the steps required to configure and manage it in development and production environments. It consists of the following sections:



Understanding the functional parts of the broker can help you configure desired broker behavior, scale operations, and optimize performance. This chapter might therefore be of more interest to administrators than to application developers.

# Broker Architecture

To perform message delivery, a broker sets up communication channels with clients, performs authentication and authorization, routes messages appropriately, guarantees reliable delivery, and provides data for monitoring system performance.

To perform this complex set of functions, a broker uses a number of different internal components, each with a specific role in the delivery process. These broker components are illustrated in Figure 4-1 and described briefly in Table 4-1. The Message Router component performs the key message routing and delivery service, and the others provide important support services on which the Message Router depends.

**Figure 4-1** Broker Components

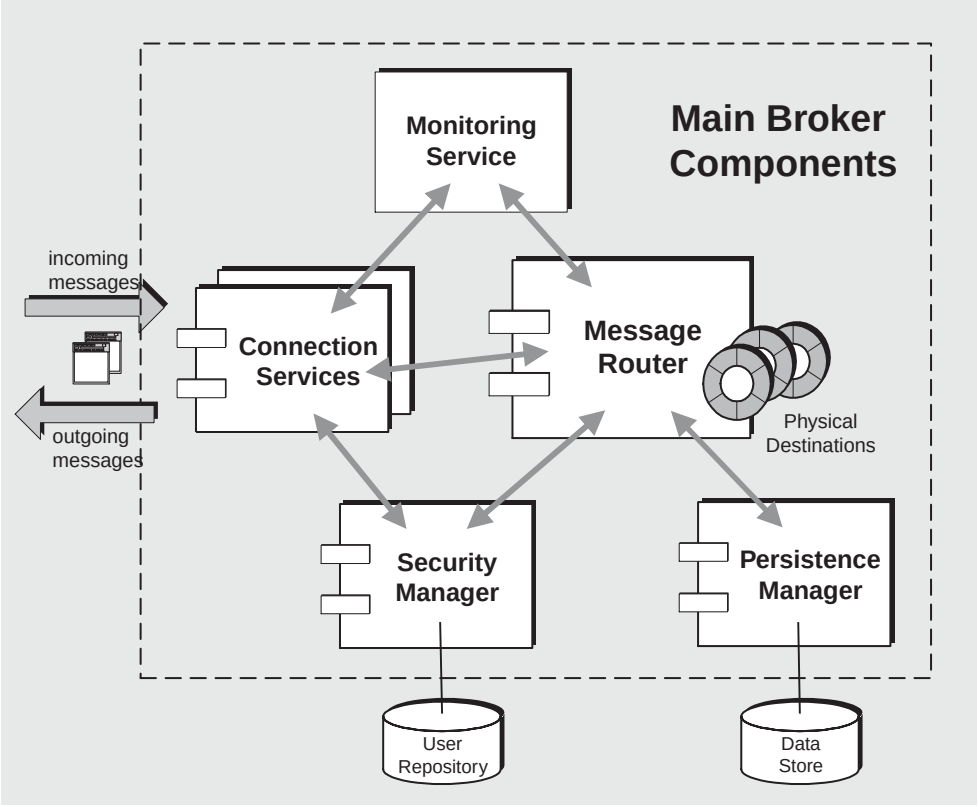


**Table 4-1** Main Broker Service Components and Functions

| Component | Description/Function |
|---|---|
| Connection Services | Manages the physical connections between a broker and clients, providing transport for incoming and outgoing messages. |
| Message Router | Manages the routing and delivery of messages: These include JMS messages as well as control messages used by the Message Queue messaging system to support JMS message delivery. |
| Persistence Manager | Manages the writing of data to persistent storage so that system failure does not result in failure to deliver JMS messages. |
| Security Manager | Provides authentication services for users requesting connections to a broker and authorization services (access control) for authenticated users. |
| Monitoring Service | Generates metric and diagnostic information that can be written to a number of output channels that you can use to monitor and manage a broker. |

When you configure the broker, you are actually configuring these services to optimize the broker's performance, depending on load conditions, application complexity, and so forth.

# Broker Components

The following sections describe each of the broker components shown in Figure 4-1, their functions, and their behavior. See the *Message Queue Administration Guide* for their respective properties and configuration procedures.

## Connection Services

A Message Queue broker supports communication with both application clients and administration clients. Each connection service is specified by its service type and protocol type.

**service type** Specifies whether the service provides JMS message delivery (`NORMAL`) or Message Queue administration services (`ADMIN`) that support administration tools.

**protocol type** Specifies the underlying transport protocol layer that supports the service.

The connection services currently available from a Message Queue broker are shown in Table 4-2.

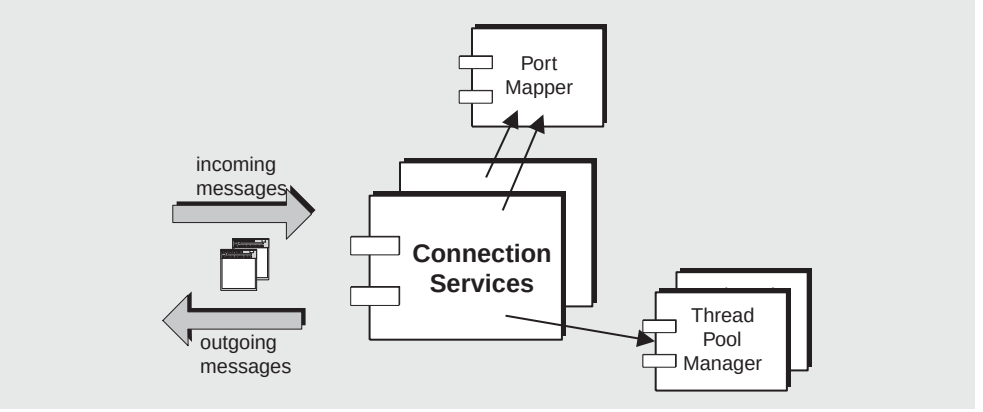
**Table 4-2** Connection Services Supported by a Broker

| Service Name | Service Type | Protocol Type |
|---|---|---|
| jms | NORMAL | TCP |
| ssljms | NORMAL | TLS (SSL-based security) |
| httpjms (Enterprise Edition) | NORMAL | HTTP |
| httpsjms (Enterprise Edition) | NORMAL | HTTPS (SSL-based security) |
| admin | ADMIN | TCP |
| ssladmin | ADMIN | TLS (SSL-based security) |

You can configure a broker to run any or all of these connection services. Each connection service supports specific authentication and authorization (access control) features (see "Security Manager" on page 82). Each connection service is multi-threaded, supporting multiple connections.

Each connection service is available at a particular port, specified by the broker's host name and a port number. The port can be dynamically allocated or you can specify the port at which a connection service is available. The general scheme is shown in Figure 4-2.

**Figure 4-2** Connection Services Support




## Port Mapper

Connection services are assigned a port by a common *Port Mapper*. The Port Mapper itself resides at a standard port number, 7676. When the Message Queue client runtime sets up a connection with the broker, it first contacts the Port Mapper, requesting the port number of the connection service it desires.

You can override the Port Mapper by assigning a *static* port number for the `jms`, `ssljms`, `admin` and `ssladmin` connection services when configuring these connection services. However, static ports are generally used only in special situations, such as in making connections through a firewall (see "HTTP/HTTPS Support" on page 76), and are not generally recommended.

## Thread Pool Manager

Each connection service is multi-threaded, supporting multiple connections. The threads needed for these connections are maintained in a thread pool managed by a *Thread Pool Manager* component. You can configure the Thread Pool Manager to set a minimum number and maximum number of threads maintained in the thread pool. As threads are needed by connections, they are added to the thread pool.

When the minimum number is exceeded, the system will shut down threads as they become free until the minimum threshold is reached, thereby saving on memory resources. The threads in a thread pool can either be dedicated to a single connection or assigned to multiple connections, as needed.

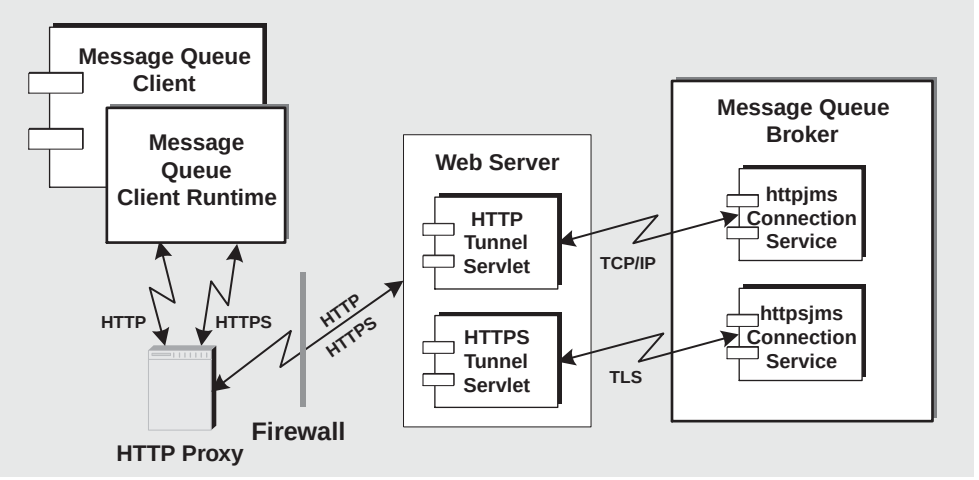
## HTTP/HTTPS Support

HTTP/HTTPS support allows Message Queue clients to interact with the broker using the HTTP protocol instead of through direct TCP connections. If the client needs to be separated from the broker by a firewall, you can use HTTP/HTTPS service because it allows communication through firewalls.

**NOTE** HTTP/HTTPS support is available for Java clients but not for C clients.

Figure 4-3 shows the main components involved in providing HTTP/HTTPS support.

**Figure 4-3** HTTP/HTTPS Support Architecture




- On the Message Queue client side, an HTTP or HTTPS transport driver encapsulates the JMS message into an HTTP request and makes sure that these requests are sent to the Web server in the correct sequence.
- The client can optionally use an HTTP proxy server to interact with the Web server, if necessary.

- An HTTP or HTTPS tunnel servlet (both bundled with Message Queue) is loaded in the Web server and used to pull JMS messages out of client HTTP requests before forwarding them to the broker. The HTTP/HTTPS tunnel servlet also sends broker responses back to the client. A single HTTP/HTTPS tunnel servlet can be used to access multiple brokers.
- On the broker side, a broker connection to the tunnel servlet is established at broker startup time. As messages are sent from the HTTP or HTTPS tunnel servlet, the `httpjms` or `httpsjms` connection service, respectively, unwraps the message and submits it to the broker's Message Router component.

The architectures for HTTP and HTTPS shown in Figure 4-3 are very similar. The main difference is that, in the case of HTTPS (`httpsjms` connection service), the tunnel servlet has a secure connection to both the client and broker.

Message Queue's HTTPS tunnel servlet passes a self-signed certificate to any broker requesting a connection. The certificate is used by the broker to set up an encrypted connection to the HTTPS tunnel servlet. Once this connection is established, a secure connection between a Message Queue client and the tunnel servlet can be negotiated.

The `httpjms` and `httpsjms` services are configured using properties described in the *Message Queue Administration Guide*.

# Message Router

Once connections have been established between clients and a broker using the supported connection services, the routing and delivery of messages can proceed.

Message Queue messaging is premised on a two-phase delivery of messages: first, from a producing client to a physical destination on the broker, and second, from the destination on the broker to one or more consuming clients. The Message Router manages the process of placing arriving messages in the appropriate destination and then routing and delivering the message to the appropriate consumer(s).

This section discusses the different kinds of destinations and the management of memory resources for those destinations, individually and collectively. A discussion of the mechanisms of routing and delivering messages can be found in Chapter 3, "Reliable Message Delivery."

## Physical Destinations

Physical destinations represent locations in a broker's physical memory where incoming messages are stored before being routed to consuming clients.

Destinations fall into a number of categories, depending on how they are created and for what purpose: admin-created, auto-created, temporary, and the dead message queue.

### *Admin-Created Destinations.*

Admin-created destinations are created by an administrator using Message Queue administration tools. These correspond either to a logical destination that is created programmatically or to a destination administered object that is looked up by the client.

Because a Message Queue message server is a central hub in a messaging system, its performance and reliability are important to the success of enterprise applications. Since destinations can consume significant resources (depending on the number and size of messages they handle, and on the number and durability of the message consumers that register), they need to be managed closely to guarantee message server performance and reliability. It is therefore standard practice for an administrator to create destinations on behalf of an application, monitor the destinations, and reconfigure their resource requirements when necessary.

### *Auto-Created Destinations*

Auto-created destinations are automatically created by the broker as they are needed, without requiring the intervention of an administrator. In particular, an auto-created destination is created whenever a message consumer or message producer attempts to access a nonexistent destination. They are used when destinations need to be created dynamically: typically during a development and test cycle. You can configure a broker to enable or disable the *auto-create* capability.

When destinations are created automatically, clashes between different client applications (using the same destination name) or degraded system performance (due to the resources required to support a destination) can result. For this reason, an auto-created destination is automatically destroyed by the broker when it is no longer being used: that is, when it no longer has message consumer clients and no longer contains any messages. If a broker is restarted, it will recreate auto-created destinations only if they contain persistent messages.

### *Temporary Destinations*

Temporary destinations are explicitly created and destroyed (using the JMS API) by clients that need a destination at which to receive replies to messages sent to other clients. These destinations are maintained by the broker only for the duration of the connection for which they are created. A temporary destination cannot be destroyed by an administrator, and it cannot be destroyed by a client application as long as it is in use: that is, if it has active message consumers. Temporary destinations, unlike admin-created or auto-created destinations (that have persistent messages), are not stored persistently and are never re-created when a broker is restarted; however, they are visible to Message Queue administration tools.

### *The Dead Message Queue.*

The *dead message queue* is a specialized destination created automatically at broker startup that is used to store dead messages for diagnostic purposes. A *dead message* is one that is removed from the system for a reason other than normal processing or explicit administrative 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.

There are two ways to have messages placed in the dead message queue:

- You can configure destinations to place messages in the dead message queue rather than discarding them.
- A client developer can also set a property value when creating a message that determines whether the message should be placed in the dead message queue were it to die.

When a message is placed in the dead message queue, additional property information is written into it, providing you with information about the cause of death.

## Memory Resource Management

A message server is limited in resources: memory, CPU cycles, and so forth. As a result, depending on the use patterns of the messaging applications the broker supports, it is possible for a message server to become overwhelmed to the point where it becomes unresponsive or unstable. In particular, this can be a problem if production of messages for a destination is much faster than consumption.

The Message Router has mechanisms for managing memory resources and preventing the broker from running out of memory. It uses three levels of memory protection to keep the system operating as resources become scarce: destination message limits, system-wide limits, and system memory thresholds.

### *Destination Message Limits*

Since messages can remain in a destination for an extended period of time, memory resources can become an issue. You do not want to allocate too much memory to a destination (there is only so much system memory), nor do you want to allocate too little (messages could be rejected).

To allow for flexibility based on the load demands of each destination, you can set properties that manage memory resources 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 which of four responses are taken by the Message Router when any such limits are reached. The four limit behaviors are:

- Slowing message producers
- Throwing out the oldest messages
- Throwing out the lowest-priority messages, according to age
- Rejecting the newest messages

### *System-Wide Message Limits*

System-wide message limits constitute a second line of protection. You can specify system-wide limits that apply collectively to all destinations on a broker: the total number of messages and the memory consumed by all messages. If any of the system-wide message limits are reached, the Message Router rejects new messages.

### *System Memory Thresholds*

System memory thresholds are a third line of protection. You can specify thresholds of available system memory 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` through `yellow` and `orange` to `red`, the broker takes increasingly serious actions of the following types:

- Throwing out in-memory copies of persistent messages in the data store.
- Throttling 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.

Both of these measures degrade performance.

If system memory thresholds are reached, then you have not adequately set destination-by-destination message limits and system-wide message limits. In some situations, it is not possible for the thresholds to catch potential memory overloads in time. Hence you should not rely on this feature, but should instead configure destinations individually and collectively to optimize memory resources.

With careful monitoring and tuning, destination-based limits and behavior can be used to balance the inflow and outflow of messages so that system overload cannot occur. While these mechanisms consume overhead and can limit message throughput, they nevertheless maintain operational integrity.

## Persistence Manager

For a broker to recover in case of failure, it needs to re-create the state of its message delivery operations. This requires it to save all persistent messages, as well as essential routing and delivery information, to a data store. To recover, the broker must also be able to do the following:

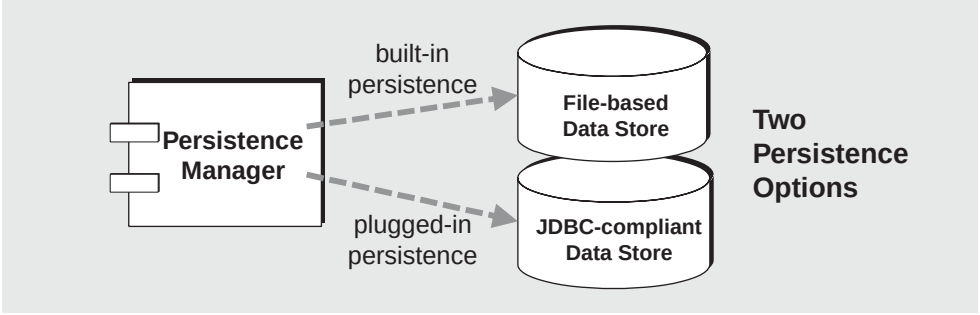
- Re-create destinations
- Restore the list of durable subscriptions for each topic
- Restore the acknowledge list for each message
- Reproduce the state of all committed transactions

The Persistence Manager manages the storage and retrieval of all this state information.

When a broker restarts, it uses the data managed by the Persistence Manager to re-create destinations and durable subscriptions, recover persistent messages, roll back open transactions, and re-create its routing table for undelivered messages. It can then resume message delivery.

Message Queue supports both built-in and plugged-in persistence modules (see Figure 4-4). Built-in persistence is a file-based data store. Plugged-in persistence uses a Java Database Connectivity (JDBC™) interface and requires a JDBC-compliant data store. The built-in persistence is generally faster than plugged-in persistence; however, some users prefer the redundancy and administrative control provided by a JDBC-compliant database system.

**Figure 4-4** Persistence Manager Support




**Built-in persistence** The default Message Queue persistent storage solution is a file-based data store that uses individual files to store persistent data. To alleviate fragmentation as messages are added and removed, you can compact the file-based data store. To maximize reliability, you can specify that persistence operations synchronize the in-memory state with the physical storage device. This helps eliminate data loss due to system crashes, but slows performance. Because the data store can contain messages of a sensitive or proprietary nature, you should secure the data store files against unauthorized access.

**Plugged-in persistence** You can set up a broker to access any data store accessible through a JDBC driver. This involves setting a number of JDBC-related broker configuration properties and using Message Queue's Database Manager utility to create a data store with the proper schema. The procedures and related configuration properties are detailed in the *Message Queue Administration Guide*.

# Security Manager

Message Queue provides authentication and authorization (access control) features, and also supports encryption capabilities:

- *Authentication* ensures that only verified users can establish a connection to a message server.
- *Authorization* specifies which users have the right to access resources like connection services or destinations to perform specific operations supported by the message service.
- *Encryption* protects messages from being tampered with during delivery over a connection.

The authentication and authorization features depend upon a user repository (see Figure 4-5 on page 84): a file, directory, or database that contains information about the users of the messaging system—their names, passwords, and *group* memberships. The names and passwords are used to authenticate a user when a connection to a broker is requested. The user names and group memberships are used, in conjunction with an access control file, to authorize operations such as producing or consuming messages for destinations.

You can populate a Message Queue-provided user repository or plug a preexisting LDAP user repository into the broker. The flat-file user repository is easy to use, but is also vulnerable to security attack, and should therefore be used only for evaluation and development purposes. The LDAP user repository is secure and therefore best suited for production purposes.

The following subsections describe authentication, authorization, and encryption. For more detailed information, see the *Message Queue Administration Guide*.

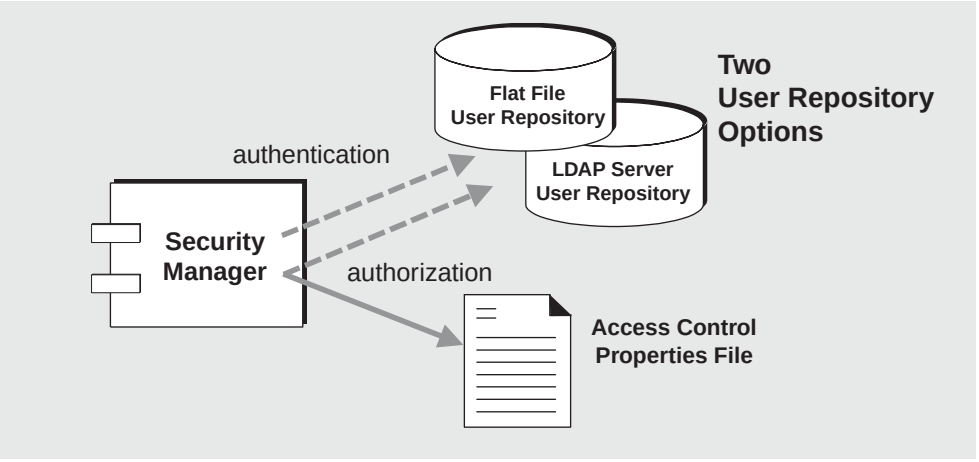
## Authentication

Message Queue security supports password-based authentication. When a client requests a connection to a broker, the client must supply a user name and password. The Security Manager compares the name and password submitted by the client 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). For more secure transmission, see "Encryption" on page 85. You can configure the type of encoding used by each connection service separately or set the encoding on a broker-wide basis.

## Authorization

Once the user of a client application has been authenticated, the user can be authorized to perform various Message Queue-related activities. The Security Manager supports both user-based and group-based access control: depending on a user's name or the groups to which the user is assigned in the user repository, that user has permission to perform certain Message Queue operations. You specify access controls in an access control properties file (see Figure 4-5).

**Figure 4-5** Security Manager Support




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

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

You can define groups and associate users with those groups in a user repository (though groups are not fully supported in the flat-file user repository). Then, by editing the access control properties file, you can specify what destinations are accessible to a user or group for production, consumption, or browsing. You can make individual destinations or all destinations accessible only to specific users or groups.

In addition, if the broker is configured to allow auto-creation of destinations (see "Auto-Created Destinations" on page 78), you can control for whom the broker can auto-create destinations by editing the access control properties file.

### 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.

## Monitoring Service

The broker includes a number of components for monitoring and diagnosing its operations. Among these are 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.

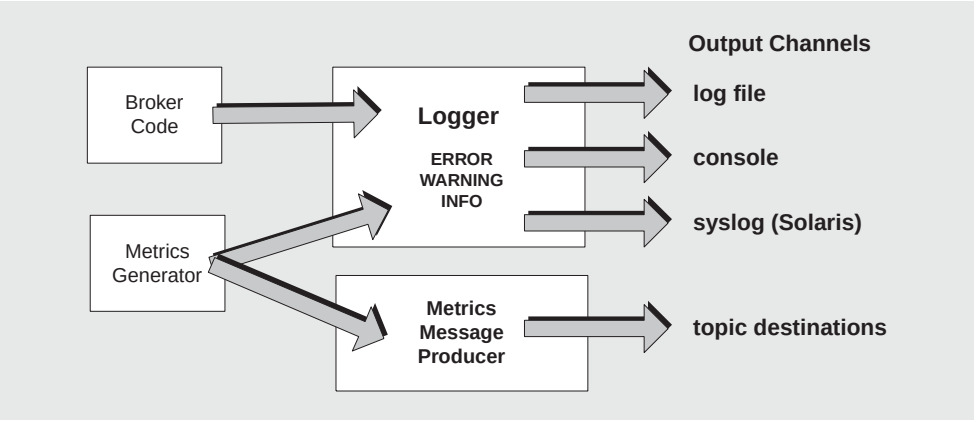
The general scheme is illustrated in Figure 4-6 on page 86.

### Metrics Generator

The metrics generator shown in Figure 4-6 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 connections open, and the number of threads being used.

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

**Figure 4-6** Monitoring Service Support




## Logger

The Message Queue Logger shown in Figure 4-6 takes information generated by broker code and the metrics generator and writes that information to a number of output channels: to standard output (the console), to a log file, and, on Solaris™ platforms, to the `syslog` daemon process.

You can 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 the *Message Queue Administration Guide*.

## Metrics Message Producer (Enterprise Edition)

The metrics Message Producer component shown in Figure 4-6 receives information from the Metrics Generator component 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 in the destinations and process the metric information 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 the *Message Queue Developer's Guide for Java Clients*, which explains how to develop a Message Queue client for consuming metrics messages. For information about how to configure the production of metrics messages, see the *Message Queue Administration Guide*.

# Development and Production Environments

The messaging infrastructure provided by the Message Queue service is needed to develop and test messaging applications, as well as to deploy and manage those applications in a production environment.

This section introduces the different approaches to using Message Queue in development and production environments. It covers the following topics:



Although the administrator has charge of setting up and managing production environments and tasks, it is important for the developer to understand such environments in order to provide administrators with the information needed to set up and configure some parts of this environment and to determine whether clients are behaving as expected.

## Development Environments and Tasks

In a development environment, the work focuses on programming Message Queue client applications. The Message Queue service is needed principally for testing.

### Out-of-the-Box Configuration

The Message Queue product is designed to be used out of the box. You can start up a broker with default values and you will be provided with a default data store, user repository, and access control properties file.

The default user repository is created with default entries that allow the Message Queue broker to be used immediately after installation without any intervention by an administrator. In other words, no initial user/password setup is required for the broker to be used. The default user name (`guest`) and password (`guest`) can be used to authenticate a client user.

A number of sample applications are also provided to guide you in developing new applications.

### Development Practices

In a development environment, the emphasis is on flexibility, and you typically adopt the following practices:

- You want minimal administration, consisting mostly of starting up a broker for developers to use in testing.
- You use default implementations of the data store (built-in file-based persistence), user repository (file-based user repository), and access control properties file. These default implementations are usually adequate for developmental testing.
- You use a simple file-system object store (by creating a directory for that purpose) in which to store administered objects, or you instantiate administered objects in client code and don't use an object store at all.
- If you are performing multiple-broker testing, you probably would not use a Master Broker (see "Cluster Synchronization" on page 94).
- You generally use auto-created destinations rather than explicitly create destinations.

## Production Environments and Tasks

In a production environment, applications must be deployed, managed, and tuned to maximize performance. In such situations the management and tuning of the messaging infrastructure is an important, if not crucial, requirement.

In addition, the deployment must support enterprise requirements, both in scale and availability. This requires customized configuration and setup of the message service, tuning performance and scaling the system to meet increasing loads, and performing day-to-day monitoring and management of both the message service and application-specific resources. The administration tasks therefore depend on the complexity of your messaging system and the complexity of the applications it must support. The following sections group administrative tasks into setup operations and maintenance operations. For the procedures required to perform these tasks, see the *Message Queue Administration Guide*.

### Setup Operations

A production environment requires setting up secure access, configuring connection factory and destination objects, setting up clusters, configuring persistent stores, and managing memory.

➤ **To Set Up a Production Environment**

Typically you have to perform at least some, if not all, of the following setup operations:

- **Secure administrative access** (protected use of administration tools):
  - Make sure `admin` connection service is activated.
  - Authorization: Allow access to `admin` connection service for a specific individual or `admin` group.
  - If authorization is for a group, make sure the administrator belongs to the admin group.
    - File-based user repository: Has a default `admin` group. Make sure administrator is in `admin` group, or if using the default `admin` user, change the `admin` password.
    - LDAP user repository: Make sure administrator is in `admin` group
- **Secure client access**:
  - Authentication: Make entries into the file-based user repository or configure the broker to use an existing LDAP user repository.

    (At a minimum, you want to password-protect administration capability.)
  - Authorization: Modify access settings in the access control properties file.
  - Encryption: Set up SSL-based connection services.
- **Create physical destinations**
- **Create administered objects**:
  - Configure or set up an LDAP object store.
  - Create connection factory and destination administered objects.
- **Create broker clusters if required**:
  - Create a central configuration file.
  - Designate a master broker.
- **Configure the broker to use plugged-in persistence.**
- **Configure memory management**

  Set destination attributes so that the number of messages and the amount of memory allocated for messages fit within available broker memory resources.

## Maintenance Operations

In a production environment, Message Queue message server resources need to be monitored and controlled. Application performance, reliability, and security are at a premium, and you have to perform a number of ongoing tasks, described below, using Message Queue administration tools:

➤ **To Maintain a Production Environment**

- **Manage application behavior**
  - Disable the broker's auto-create capability.
  - Create physical destinations on behalf of applications.
  - Set user access to destinations.
  - Monitor and manage destinations.
  - Monitor and manage durable subscriptions.
  - Monitor and manage transactions.
- **Monitor and tune the broker**
  - Use broker metrics to tune and reconfigure the broker.
  - Manage broker memory resources.
  - Add brokers to clusters to balance loads.
  - Recover failed brokers.
- **Manage administered objects:**
  - Create additional connection factory and destination administered objects as needed.
  - Adjust connection factory attribute values to improve performance and throughput.

Chapter 5

# Broker Clusters

Message Queue Enterprise Edition supports the use of *broker clusters:* groups of brokers working together to provide message delivery services to clients. Clusters enable a message server to scale its 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:



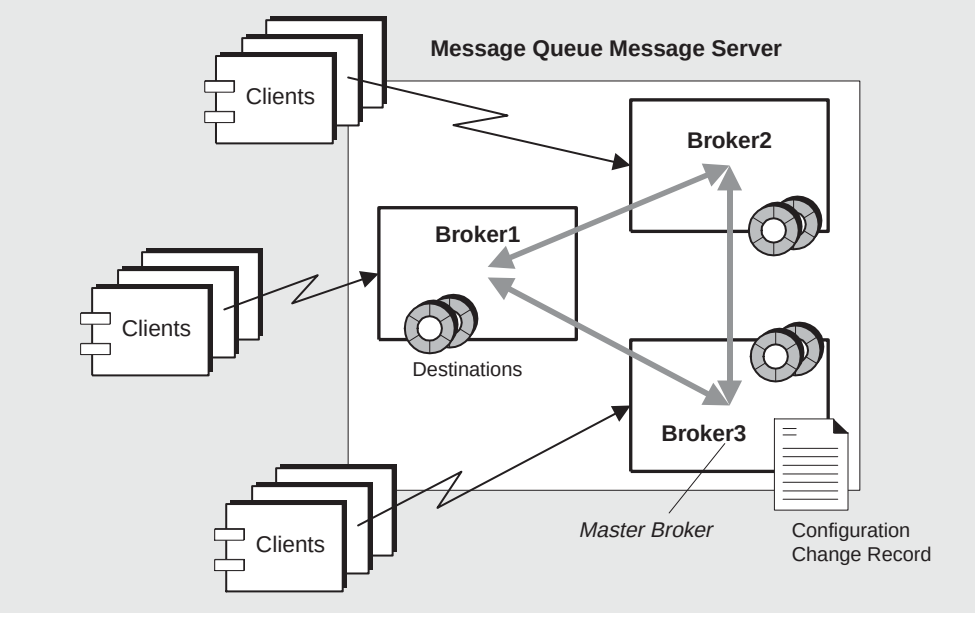
You need to read this chapter if you are an administrator charged with configuring and managing a broker cluster or a developer who needs to test a messaging application using a cluster.

# Cluster Architecture

Figure 5-1 shows Message Queue's architecture for 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 on the server. Behind the scenes, the home broker works in concert with the other brokers in the cluster to share the load of providing delivery services for all connected clients.

One broker within the cluster can be designated as the *master broker.* The master broker maintains a *configuration change record* in which changes to the cluster's persistent entities (destinations and durable subscriptions) are recorded. This record is used to propagate such change information to brokers that were offline at the time the changes occurred; see "Cluster Synchronization," below, for further discussion.

**Figure 5-1** Cluster Architecture




The following sections discuss how message delivery takes place within a cluster and how the brokers are configured and synchronized, even in the case when one or more has been offline.

## Message Delivery

In a cluster configuration, each destination is replicated on all brokers in the cluster. Each broker knows about message consumers that are registered for destinations on all other brokers. Each broker can therefore route messages from its own directly connected message producers to remote message consumers, and can deliver messages from remote producers to its own directly connected consumers.

A message producer's own home broker handles all storage and routing, and processes all client acknowledgments, for messages originated by that producer. To minimize message traffic within the cluster, messages are sent from one broker to another only when they are to be delivered to a consumer connected to the target broker. In some cases (such as queue delivery to multiple consumers), traffic can be further reduced by specifying that delivery to local consumers have priority over delivery to remote consumers. In situations requiring secure, encrypted message delivery between client and message server, a cluster can also be configured to provide secure delivery of messages between brokers.

**NOTE** With some exceptions, destination properties in a cluster apply collectively to the cluster as a whole, rather than to individual destination instances. See the *Message Queue Administration Guide* for information on specific destination properties.

## Cluster Configuration

To establish connections between the brokers in a cluster at startup time, each broker must be passed the host names and port numbers of all the others (including the master broker, if any). This information is specified by a set of *cluster configuration properties,* which should be uniform for all brokers in the cluster. Although you can specify the configuration properties for each broker individually, this approach is error-prone and can easily lead to inconsistencies in the cluster configuration. Instead, it is recommended that you place all the configuration properties in one central *cluster configuration file* that is referenced by each broker at startup time. This ensures that all brokers share the same configuration information.

See the *Message Queue Administration Guide* for detailed information on cluster configuration properties.

**NOTE** Although the cluster configuration file was originally intended for configuration purposes, it is also a convenient place to store other 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. Such configuration changes include the following:

- A destination on one of the cluster's brokers is created or destroyed.
- 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.

Such configuration change information 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. To accommodate offline brokers, Message Queue 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 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.)

# Deployment Environment

The use of broker clusters depends on whether they are deployed in a development environment or a production environment.

## Development Environments

In development environments, where a cluster is used for testing and where scalability and broker recovery are not important considerations, there is little need for a master broker. Under test conditions, destinations are often auto-created (see "Auto-Created Destinations" on page 78) and durable subscriptions to these destinations are created and destroyed by the applications being tested. In the absence of a master broker, Message Queue relaxes the requirement that a master broker be running in order to start other brokers, and allows changes in destinations and durable subscriptions to be made and propagated to all running brokers. (If a broker goes offline and is subsequently restored, however, it will not synchronize with changes made while it was offline.) If you reconfigure the environment to use a master broker, Message Queue will reimpose the normal requirements.

## Production Environments

In production environments, where scalability and broker recovery *are* important considerations, it is essential to use a master broker and maintain the configuration change record. This guarantees that if a broker goes offline and is subsequently restored, it will synchronize with changes made while it was offline.

In fact, it is a good idea to make a periodic backup of the configuration change record to guard against accidental corruption of the record and safeguard against failure of the master broker. Message Queue provides command-line options for backing up and restoring the configuration change record. If necessary, you can also change the broker serving as the master broker. See the *Message Queue Administration Guide* for more information.

Chapter 6

# 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 with one another through reliable, asynchronous message exchange. 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:



Because this chapter covers both programming and deploying J2EE components, it is of interest to both application developers and administrators.

# JMS/J2EE Programming: Message-Driven Beans

In addition to the general JMS client programming model introduced in "JMS Programming Model" on page 28, there is a more specialized adaptation of JMS used in the context of J2EE platform applications. This specialized JMS client is called a *message-driven bean* and is one of a family of Enterprise JavaBeans (EJB) components described in the EJB 2.0 Specification (`http://java.sun.com/products/ejb/docs.html`).

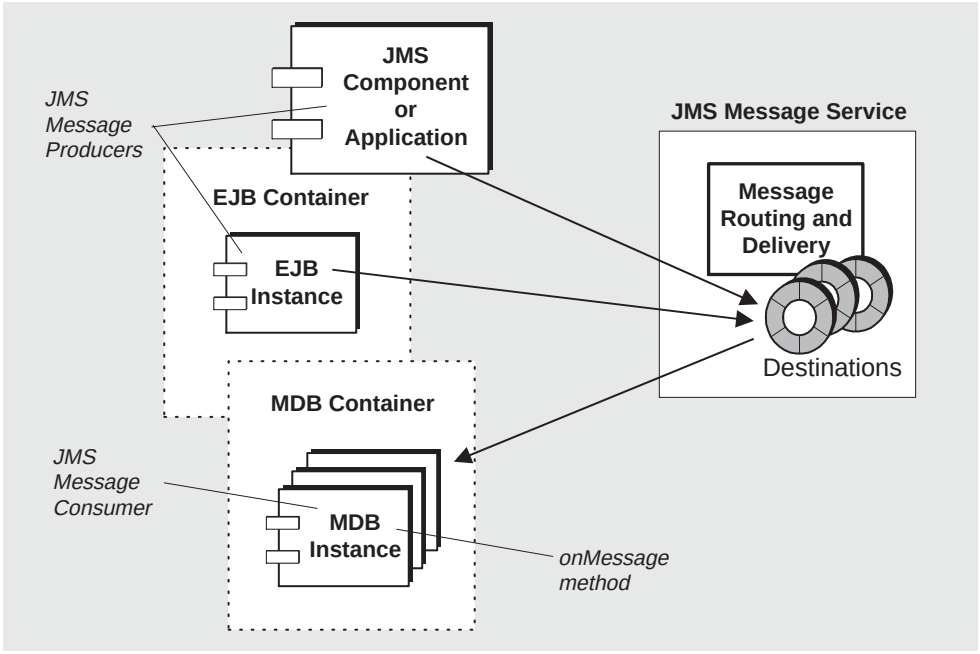
The need for message-driven beans arises out of the fact that other EJB components (session beans and entity beans) can only be called synchronously. These EJB components have no mechanism for receiving messages asynchronously, since they are accessed only through standard EJB interfaces.

However, asynchronous messaging is a requirement of many enterprise applications. Most such applications require that server-side components be able to communicate and respond to each other without tying up server resources. Hence the need for an EJB component that can receive messages and consume them without being tightly coupled to the producer of the message. This capability is needed for any application in which server-side components must respond to application events. In enterprise applications, this capability must also scale under increasing load.

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

**Message-driven Bean** The MDB is a JMS message consumer that implements the JMS `MessageListener` 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 6-1.

**Figure 6-1** Messaging with MDBs




**MDB Container** The MDB is supported by a specialized EJB container, responsible for creating instances of the MDB and setting them up for asynchronous consumption of messages. This involves setting up a connection with the message service (including authentication), creating a pool of sessions associated with a given destination, and managing the distribution of messages as they are received among the pool of sessions and associated MDB instances. Since the container controls the life cycle of MDB instances, it manages the pool of MDB instances so as to accommodate incoming message loads.

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

# J2EE Application Server Support

In J2EE architecture (see the J2EE Platform Specification located at http://java.sun.com/j2ee/download.html#platformspec), 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.

# 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 is defined by the J2EE Connector Architecture (J2EECA) 1.5 specification.) This architecture allows any application server (that complies with J2EE 1.4) to interact with external systems in a standard way. External systems can include various enterprise information systems (EIS), as well as various 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.

The standard interactions facilitated by J2EECA 1.5 include connection pooling, thread pooling, transaction and security context propagation, as well as support for message-driven bean containers of various kinds. The specification also includes a standard way to create connection factories and other administered objects.

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 message server 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 the *Message Queue 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 becomes certified for those application servers, Message Queue documentation will provide specific information on the relevant deployment and configuration procedures.

Appendix A

# 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.

This material is most relevant to application developers.

The handling of each of the JMS optional items is indicated in Table A-1:

**Table A-1** Optional JMS Functionality

| Section in JMS Specification | Description and Message Queue Handling |
|---|---|
| 3.4.3<br>`JMSMessageID` | "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."<br><br>**Message Queue implementation**: Product does not disable Message ID generation (any `setDisableMessageID`() call in `MessageProducer` is ignored). All messages will contain a valid `MessageID` value. |
| 3.4.12<br>Overriding Message Header Fields | "JMS does not define specifically how an administrator overrides these header field values. A JMS provider is not required to support this administrative option."<br><br>**Message Queue implementation**: The Message Queue product supports administrative override of the values in message header fields through configuration of the client runtime (see "Overriding Message Header Values" on page 41). |

**Table A-1** Optional JMS Functionality *(Continued)*

| Section in JMS Specification | Description and Message Queue Handling |
|---|---|
| 3.5.9<br>JMS Defined Properties | "JMS Reserves the '`JMSX`' Property name prefix for JMS defined properties."<br>"Unless noted otherwise, support for these properties is optional." |
| | **Message Queue implementation**: The `JMSX` properties defined by the JMS 1.1 specification are supported in the Message Queue product (see the *Message Queue Administration Guide*). |
| 3.5.10<br>Provider-specific Properties | "JMS reserves the '`JMS_`<*vendor_name*>' property name prefix for provider-specific properties." |
| | **Message Queue implementation**: 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. Message Queue 3 does not use provider-specific properties. |
| 4.4.8<br>Distributed Transactions | "JMS does not require that a provider support distributed transactions." |
| | **Message Queue implementation**: Distributed transactions are supported in this release of the Message Queue product (see "Transactions" on page 65). |
| 4.4.9<br>Multiple Sessions | "For PTP <point-to-point distribution model>, JMS does not specify the semantics of concurrent `QueueReceivers` 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. |
| | **Message Queue implementation**: The Message Queue implementation supports queue delivery to multiple consumers. For more information, see "Queue Delivery to Multiple Consumers" on page 61. |

# Glossary

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

**acknowledgement** Control messages exchanged between clients and message server to ensure reliable delivery. There are two general types of acknowledgement: client acknowledgements and broker acknowledgements.

**administered objects** 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.

**asynchronous messaging** 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.

**authentication** The process by which only verified users are allowed to set up a connection to a message server.

**authorization** 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.

**broker** 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.

**client** 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.

**client identifier** An identifier that associates a connection and its objects with a state maintained by the Message Queue message server on behalf of the client.

**client runtime** Message Queue software that provides messaging clients with an interface to the Message Queue message server. The client runtime supports all operations needed for clients to send messages to destinations and to receive messages from destinations. The client runtime is configured by setting `ConnectionFactory` properties.

**cluster** Two or more interconnected brokers that work in concert to provide scalable messaging services.

**connection** A communication channel between a client and a message server used to pass both payload messages and control messages.

**connection factory** The administered object the client uses to create a connection to a message server. This can be a `ConnectionFactory` object, a `QueueConnectionFactory` object or a `TopicConnectionFactory` object.

**consumer** An object (MessageConsumer) 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 (`QueueReceiver` or `QueueBrowser`); in the publish/subscribe delivery model, the consumer is a subscriber (`TopicSubscriber`).

**data store** A database where information (durable subscriptions, data about destinations, persistent messages, auditing data) needed by the broker is permanently stored.

**dead message** 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.

**dead message queue** A specialized destination created automatically at broker startup that is used to store dead messages for diagnostic purposes.

**delivery mode** 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).

**delivery model** 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.

**destination** The physical destination in a Message Queue message server 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.

**domain** 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.

**encryption** A mechanism for protecting messages from being tampered with during delivery over a connection.

**group** The group to which the user of a Message Queue client belongs for purposes of authorizing access to connections, destinations, and specific operations.

**JMS provider** 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.

**message server** One or more brokers that provide centralized delivery services for the Message Queue service, including connections to clients, message handling and routing, persistence, security, and monitoring. The message server maintains physical destinations to which producing clients send messages, and from which the messages are delivered to consuming clients.

**message service** A middleware service that provides asynchronous, reliable exchange of messages between distributed components or applications. It includes a message server, the client runtime, and the several data stores needed by the message server to carry out its functions.

**messages** 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.

**messaging** A system of asynchronous requests, reports, or events used by enterprise applications that allows loosely coupled applications to transfer information reliably and securely.

**producer** An object (MessageProducer) 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 (`QueueSender`); in the publish/subscribe delivery model, a producer is a publisher (`TopicPublisher`).

**queue** 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.

**selector** 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.

**session** A single threaded context for sending and receiving messages. This can be a queue session or a topic session.

**topic** 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.

**transaction** An atomic unit of work that must either be completed or entirely rolled back.

# Index

# C

## D



## E



## F

# H



# I



# J



# L



# M

## N



## O

# P



# Q



# R



# S

## T



## U



## W



## X