Sun Java System Message Queue 4.2 Technical Overview

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

Part No: 820–4917
Ma;y 2008

Contents

Preface

This book, the *Sun Java System Message Queue 3.7 UR1Technical Overview*, provides an introduction to the technology, concepts, architecture, capabilities, and features of the Message Queue messaging service.

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

Who Should Use This Book

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

- An *application developer* is responsible for writing Message Queue client applications that use the Message Queue service to exchange messages with other client applications.
- An *administrator* is responsible for setting up and managing a Message Queue messaging service.

This book does not assume any knowledge of messaging systems or 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 and technology before reading the Message Queue developer and administration guides.

How This Book Is Organized

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

TABLE P–1 Book Contents and Organization

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

Related Documentation

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

TABLE P–2 Message Queue Documentation Set

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

Online Help

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

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

JavaDoc

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

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

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

Example Client Applications

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

Example Java Client Applications

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

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

Example C Client Programs

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

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

Platform	Location
Windows	`IMQ_HOME/demo/C/`

The Java Message Service (JMS) Specification

The JMS specification can be found at the following location:

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

The specification includes sample client code.

Directory Variable Conventions

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

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

TABLE P–3 Directory Variable Conventions

Variable	Description
`IMQ_HOME`	Used in Message Queue documentation to refer to the Message Queue base directory (root installation directory): ■ On Solaris and Linux, there is no root Message Queue installation directory: files are installed in several different directories. Therefore `IMQ_HOME` is not used in Message Queue documentation to refer to file locations in Solaris and Linux. ■ On Windows, the root Message Queue installation directory is set to the directory in which you unzip the Message Queue bundle.

TABLE P–3 Directory Variable Conventions *(Continued)*

Variable	Description
IMQ_VARHOME	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. ■ On Solaris, IMQ_VARHOME defaults to the /var/imq directory. ■ On Solaris, for Sun Java System Application Server, Evaluation Edition, IMQ_VARHOME defaults to the IMQ_HOME/var directory. ■ On Linux, IMQ_VARHOME defaults to the /var/opt/sun/mq directory. ■ On Windows, IMQ_VARHOME defaults to the IMQ_HOME/var directory.
IMQ_JAVAHOME	An environment variable that points to the location of the Java runtime environment (JRE) required by Message Queue executables: ■ On Solaris, IMQ_JAVAHOME looks for the latest JDK, but a user can optionally set the value to wherever the preferred JRE resides. ■ On Linux, Message Queue first looks for the latest JDK, but a user can optionally set the value of IMQ_JAVAHOME to wherever the preferred JRE resides. ■ On Windows, IMQ_JAVAHOME will be set to point to an existing Java runtime if a supported version is found on the system. If a supported version is not found, one will be installed.

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

Typographic Conventions

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

TABLE P–4 Typographic Conventions

Typeface	Meaning	Example
AaBbCc123	The names of commands, files, and directories, and onscreen computer output	Edit your .login file. Use ls -a to list all files. machine_name% you have mail.
AaBbCc123	What you type, contrasted with onscreen computer output	machine_name% **su** Password:

TABLE P–4 Typographic Conventions *(Continued)*

Typeface	Meaning	Example
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

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

TABLE P–5 Shell Prompts

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

Symbol Conventions

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

TABLE P–6 Symbol Conventions

Symbol	Description	Example	Meaning
`[ ]`	Contains optional arguments and command options.	`ls [-l]`	The `-l` option is not required.

TABLE P–6 Symbol Conventions *(Continued)*

Symbol	Description	Example	Meaning
`{ | }`	Contains a set of choices for a required command option.	`-d {y|n}`	The `-d` option requires that you use either the `y` argument or the `n` argument.
`${ }`	Indicates a variable reference.	`${com.sun.javaRoot}`	References the value of the `com.sun.javaRoot` variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

Documentation, Support, and Training

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

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

Searching Sun Product Documentation

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

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

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

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

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

Third-Party Web Site References

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

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

Sun Welcomes Your Comments

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

◆ ◆ ◆ CHAPTER 1

Messaging Systems: An Introduction

Sun Java™ System Message Queue is a messaging middleware product that implements and extends the Java Message Service (JMS) standard. If you are familiar with the JMS specification, you can skip to the section of this chapter on "Message Queue: Elements and Features" on page 26. Otherwise, you should begin at the beginning.

This chapter describes the messaging technology that underlies Message Queue and explains how Message Queue implements and extends the JMS specification. The chapter covers the following topics:

Message-Oriented Middleware (MOM)

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

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

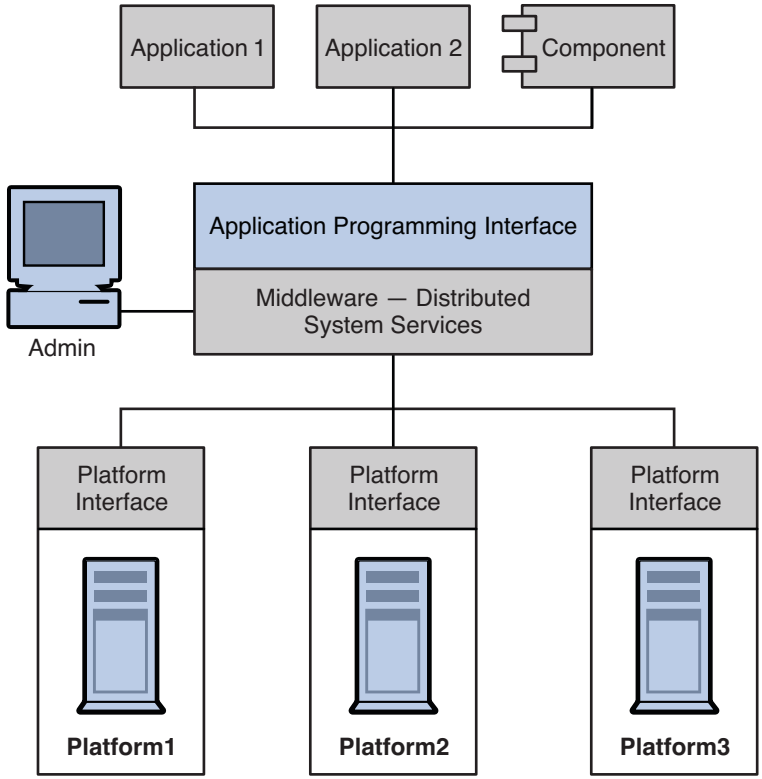

FIGURE 1–1 Middleware

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

Middleware can be grouped into the following categories:

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

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

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

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

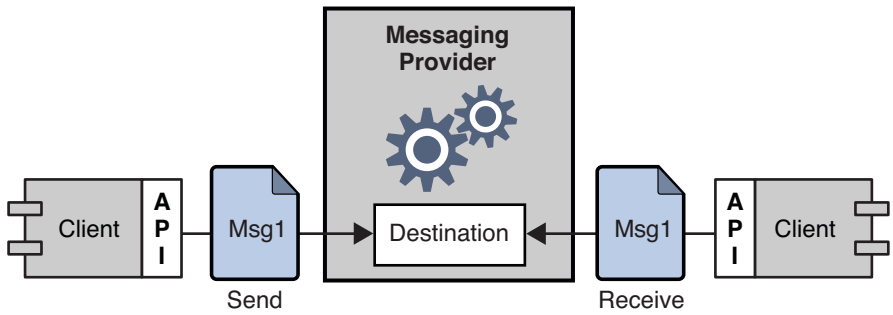

FIGURE 1–2 MOM-Based System

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

Using a MOM system, a client makes an API call to send a message to a *destination* managed by the provider. The call invokes provider services to route and deliver the message. Once it has sent the message, the client can continue to do other work, confident that the provider retains the message until a receiving client retrieves it. The message-based model, coupled with the mediation of the provider, makes it possible to create a system of loosely-coupled components. Such a system can contin

One other advantage of having a messaging provider mediate messaging between clients is that by adding an administrative interface, you can monitor and tune performance. Client

applications are thus effectively relieved of every problem except that of sending, receiving, and processing messages. It is up to the code that implements the MOM system and up to the administrator to resolve issues like interoperability, reliability, security, scalability, and performance.

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

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

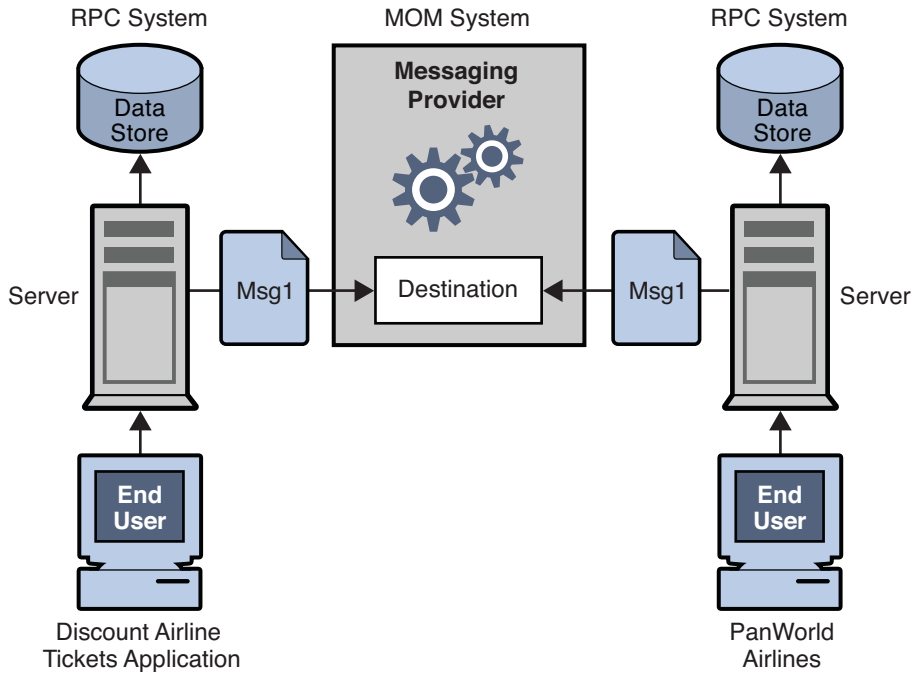

FIGURE 1–3 Combining RPC and MOM Systems

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

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

JMS as a MOM Standard

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

In creating the JMS specification, its designers wanted to capture the following essential elements of existing messaging systems:

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

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

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

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

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

JMS Messaging Objects and Domains

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

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

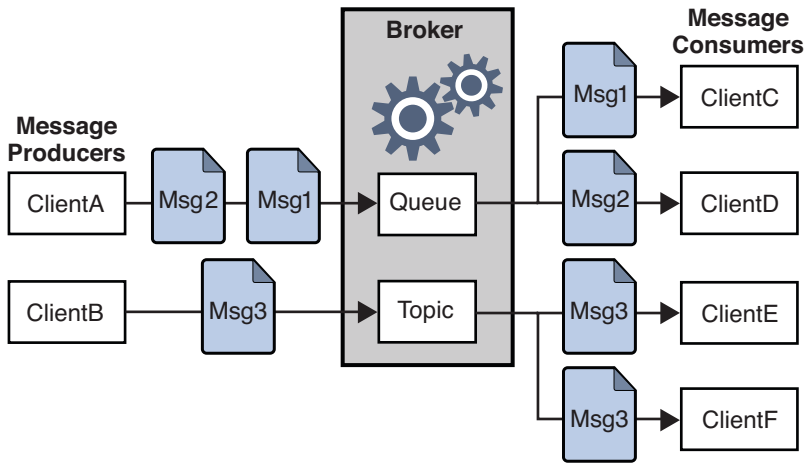

FIGURE 1–4 JMS Messaging Domains

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

- Messaging between clients A, C, and D illustrates the *point-to-point* domain. Using this pattern, a client sends a message to a queue destination from which only one receiver may get it. No other receiver accessing that destination can get that message.

- Messaging between clients B, E, and F illustrates the *publish/subscribe* domain. Using this broadcast pattern, a client sends a message to a topic destination from which any number of consuming subscribers can retrieve it. Each subscriber gets its own copy of the message.

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

Administered Objects

The JMS specification created a standard that combined many elements of existing MOM systems without attempting to exhaust all possibilities. Rather, it sought to set up an extensible scheme that could accommodate differences and future growth. JMS leaves a number of messaging elements up to the individual JMS providers to define and implement. These include load balancing, standard error messages, administrative APIs, security, the underlying wire protocols, and message stores. The next section, "Message Queue: Elements and Features" on page 26 describes how Message Queue implements many of these elements and how it extends the JMS specification.

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

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

JMS clients, however, are not required to look up administered objects; they can create these objects programmatically. For quick prototyping, creating these objects programmatically might be easiest. But for deployment in a production environment, looking up administered objects in a central repository makes it much easier to control and manage messaging behavior throughout the system:

- By using administered objects for connection factory objects, administrators can tune messaging performance by reconfiguring these objects. Performance can be improved without having to recode client applications.
- By using administered objects for physical destinations, administrators can control the proliferation of these destinations (which can be auto-created) on the broker by requiring clients to access only preconfigured destination objects.
- Administered objects shield client developers from provider-specific implementation details and allow the code they develop for one provider to be portable to other providers with little or no change.

The use of administered objects completes the set of elements in a JMS application system, as shown in Figure 1–5.

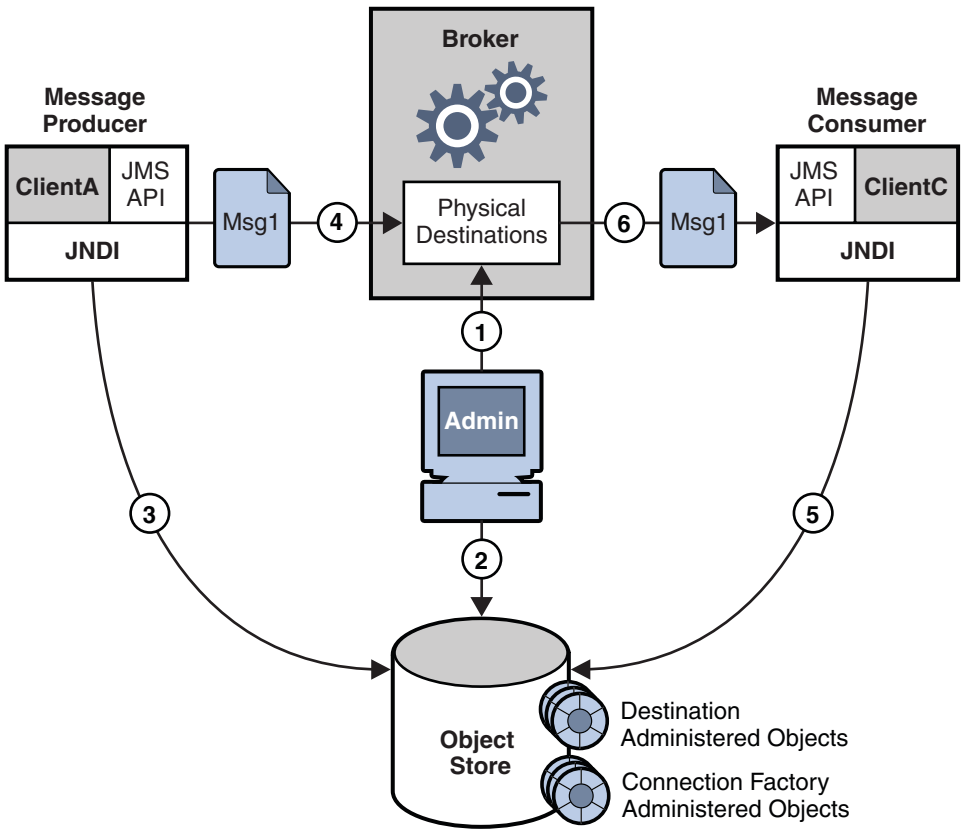

FIGURE 1–5 Basic Elements of a JMS Application System

Figure 1–5 shows how a message producer and a message consumer use destination administered objects to access the physical destination to which they correspond. The marked steps denote the actions that need to be taken by the administrator and by the client applications to send and receive messages using this mechanism:

1. The administrator creates a physical destination on the broker.
2. The administrator creates a destination administered object and configures it by specifying the name of the physical destination to which it corresponds and its type: queue or topic.
3. The message producer uses a JNDI call to look up the destination administered object that points to the corresponding physical destination.
4. The message producer sends a message to the physical destination.
5. The message consumer uses a JNDI call to look up the destination administered object that points to the corresponding physical destination from which it expects to get messages.
6. The message consumer gets the message from the physical destination.

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

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

Message Queue: Elements and Features

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

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

This section covers the following topics:

The Message Queue Service

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

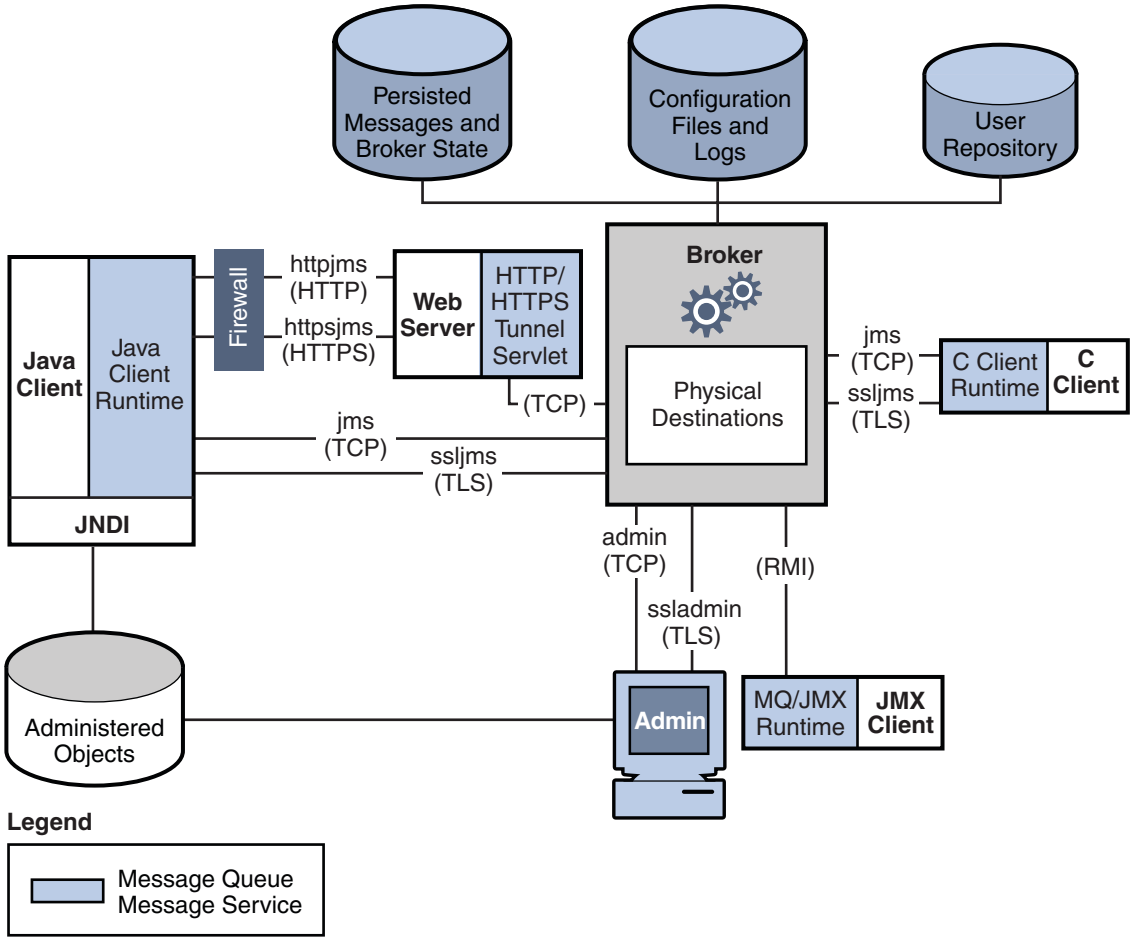

FIGURE 1–6 Message Queue Service

As you can see, a full-featured JMS provider is more complex than the basic JMS model might lead one to suspect. The following sections introduce the elements of the Message Queue service shown in Figure 1–6::

The Broker

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

- To route and deliver messages, the broker places incoming messages in their respective destinations and manages message flow into and out of these destinations.
- To provide reliable delivery, the broker uses a persistent store to save state information and persistent messages until they are received. Should the broker or the connection fail, the saved information allows the broker to restore the broker's state and to retry operations.
- To provide security for the data being exchanged the broker uses authenticated connections. Optionally data may be encrypted by running over a secure protocol like SSL. The broker also uses and manages a repository that holds information about users and the data or operations they can access. The broker authenticates users who are requesting services and authorizes the operations they want to carry out by looking up information in this repository.
- To monitor the system, the broker generates metrics and diagnostic information that an administrator can access to measure performance and to tune the broker. Metrics information is also available programmatically to allow applications or administrators to adjust message flow and patterns to improve performance.

The Message Queue service provides a variety of administrative tools that the administrator can use to configure broker support. For more information, see "Built-in Administration Tools" on page 73.

Client Runtime Support

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

Java and C Client Support

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

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

There are a number of differences in the functionality provided by these two APIs; these are documented in "Java and C Clients" on page 59.

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

Support for SOAP Messages

SOAP (Simple Object Access Protocol) allows the exchange of structured data between two peers in a distributed environment. The data exchanged is specified by an XML schema. SOAP message delivery is limited to using the point-to-point domain and does not by itself guarantee reliability.

However, Message Queue Java clients are able to send and receive SOAP messages, encapsulated as JMS messages. By encapsulating a SOAP message in a JMS message and delivering it using the broker, you can take advantage of full featured Message Queue messaging, which guarantees reliable delivery and also allows you to use the publish/subscribe domain. Message Queue provides utility routines that a message producer can use to encapsulate a SOAP message as a JMS message and that a message consumer can use to extract a SOAP message from the JMS message. Message Queue also provides XML schema validation of the encapsulated XML message.

"Working with SOAP Messages" on page 58 gives you a more detailed view of SOAP message processing.

Connection Services

As shown in Figure 1–6 both application clients and administration clients can connect to the broker. The JMS specification does not dictate that providers implement any specific wire protocols. Message Queue connection services, used by application clients and administration clients to connect to the broker, are currently layered on top of TCP, TLS, HTTP, or HTTPS protocols. (Services layered on top of HTTP allow messages to pass through firewalls.). There are two general types of connection services:

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

Each connection service supports specific authentication and authorization (access control) features and each service is multi-threaded, supporting multiple connections.

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

The connections provided by MQ connection services can be configured to specify which brokers to connect to, how to handle reconnection, message flow control, and so on. For additional information about how connections can be configured, see "Connection Factories and Connections" on page 45.

Administration

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

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

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

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

In addition, to these built-in administration tools, Message Queue also supports the Java Management Extensions (JMX) specification for configuring and monitoring brokers, destinations, connection services, and so forth. Using the JMX Administration API, you can perform these administration functions programmatically from within a Java application.

Broker Clusters: Scalability and Availability

Message Queue brokers can be connected into a *broker cluster*: a set of brokers that work collectively to perform message delivery between message producers and consumers. Broker clusters add scalability and availability to the Message Queue service, as described briefly in the following sections:

- "Message Service Scalability" on page 32
- "Message Service Availability" on page 33

For additional information on broker clusters, see Chapter 4, "Broker Clusters"

Message Service Scalability

As the number of clients or the number of connections grows, you may need to scale a message service to eliminate bottlenecks or to improve performance. In general, you can scale a message service both vertically (increasing the number of client applications that are supported by a single broker) and horizontally (distributing client applications among a number of interconnected brokers).

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

Horizontal scaling is generally achieved using a broker cluster. While it is possible to scale horizontally by simply redistribute clients among additional brokers, this approach is appropriate only if your messaging operations can be divided into independent work groups. However, if producer clients must produce messages to be consumed by consumer clients connected to remote brokers, then brokers must work collectively, as part of a broker cluster, to achieve horizontal scaling.

In a broker cluster, each broker is connected to every other broker in the cluster. Brokers can reside on the same host, but more often are distributed across a network. Each broker can route messages from producers to which it is directly connected to consumers that are connected to remote brokers in the cluster.

Note – If you are using the point-to-point domain, you can scale the consumer side by allowing multiple consumers to access a queue. This is a Message Queue feature (the JMS specification defines messaging behavior in the case of only one consumer accessing a queue). When multiple consumers access a queue, the load-balancing among them takes into account each consumer's capacity and message processing rate.

Message Service Availability

In addition to providing for message service scalability, broker clusters also provide for message service availability. If one broker in a cluster fails, then other brokers in the cluster are available to continue to provide messaging services to client applications.

Message Queue supports two clustering models that provide different degrees of availability:

- **Conventional broker clusters.** A conventional broker cluster provides *message service availability*. When a broker or a connection fails, clients connected to the failed broker reconnect to another broker in the cluster. However, messages and state information stored in the failed broker cannot be recovered until the failed broker is brought back online. This can result in an interruption of message delivery.
- **High-availability broker clusters.** A high-availability broker cluster provides *data availability* in addition to message service availability. When a broker or a connection fails, another broker takes over the pending work of the failed broker. The failover broker has access to the failed broker's messages and state information. Clients connected to the failed broker reconnect to the failover broker. In a high-availability cluster, as compared to a conventional cluster, a failure results in no interruption of message delivery.

Note – You can also achieve data availability in a conventional cluster by using Sun Cluster software. Sun Cluster software replicates broker data and provides for a hot standby broker to take over the pending work of a failed broker. Sun Cluster software, however, is only supported on the Solaris operating system.

Message Queue as an Enabling Technology

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

A Java EE-compliant application server must use a resource adapter furnished by a given JMS provider to use the functionality of that provider. Message Queue provides such a resource adapter. Using the support of a plugged in JMS provider, Java EE components, including

MDBs, deployed and running in the application server environment can exchange JMS messages among themselves and with external JMS components. This provides a powerful integration capability for distributed components.

For information on the Message Queue resource adapter, see Chapter 5, "Message Queue and Java EE"

Message Queue Feature Summary

Message Queue has capabilities and features that far exceed 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.

The following enterprise-strength features, which are listed alphabetically in Appendix B, "Message Queue Features", can be divided into the quality-of-service categories below:

Integration Support

- Multiple connection services, including HTTP connections and secure connections
- Java EE resource adapters
- SOAP support
- Schema validation of XML messages
- C client support
- C-API support for distributed transactions
- LDAP server support

Security

- Authentication
- Authorization, including JAAS-based authentication
- Secure connections, including encryption

Scalability

- Broker clusters
- Queue delivery to multiple consumers
- Thread management
- Multiple destinations for publishers and subscribers

Availability

- Broker clusters, including conventional clusters and high-availability clusters
- Connection ping
- Automatic reconnect
- Connection event notification

Performance

- Tunable performance
- Memory resource management
- Message flow control
- Configurable physical destinations
- Message compression

Serviceability

- Administration tools
- Message-based monitoring API
- JMX-based administration
- Java ES Monitoring Framework support
- Client runtime logging
- Dead message queue
- Broker configurations
- Configurable persistence
- Support for Sun Connection Registration

◆ ◆ ◆ CHAPTER 2

Client Programming Model

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

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

Messaging Domains

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

Point-To-Point Messaging

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

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

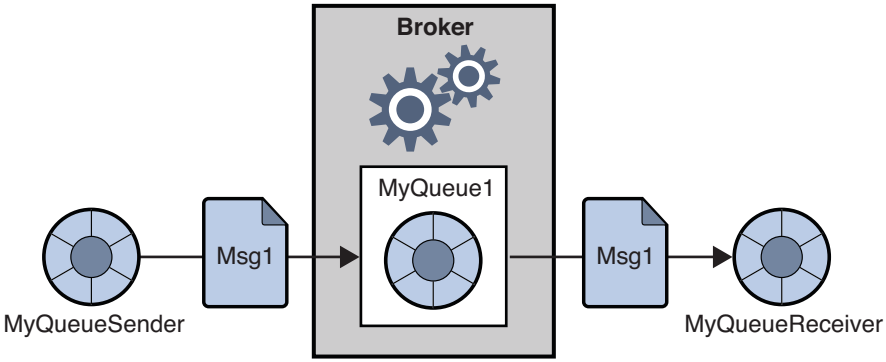

FIGURE 2–1 Simple Point-to-Point Messaging

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

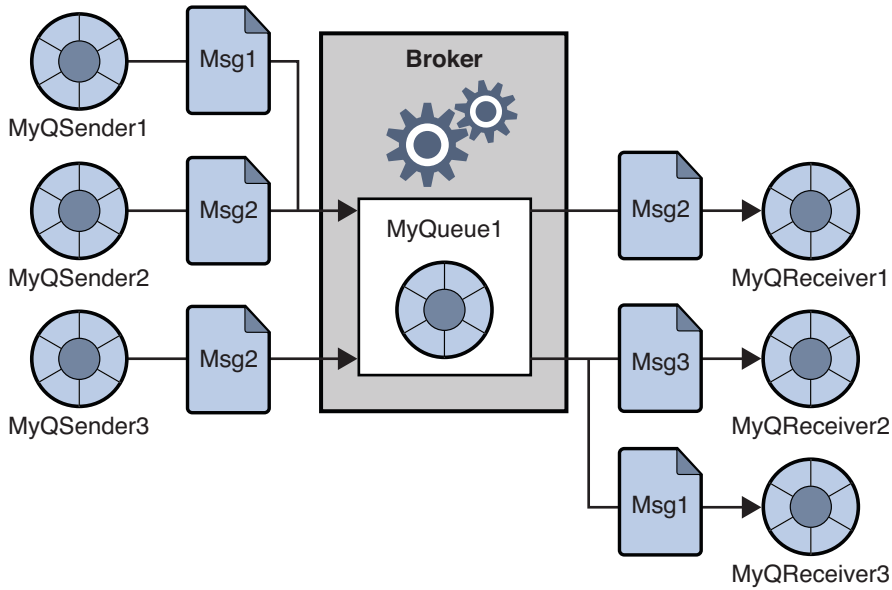

FIGURE 2–2 Complex Point-to-Point Messaging

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

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

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

- Messages destined for a queue are always retained, even if there are no receivers.

- Java clients can use a *queue browser* object to inspect the contents of a queue. They can then consume messages based on the information gained from this inspection. That is, although the consumption model is normally FIFO (first in, first out), receivers can consume messages that are not at the head of the queue by using message selectors. Administrative clients can also use the queue browser to monitor the contents of a queue.
- The fact that multiple receivers can consume messages from the same queue allows you to use load-balancing to scale message consumption if the order in which messages are received is not important. (This is a Message Queue extension.)

Publish/Subscribe Messaging

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

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

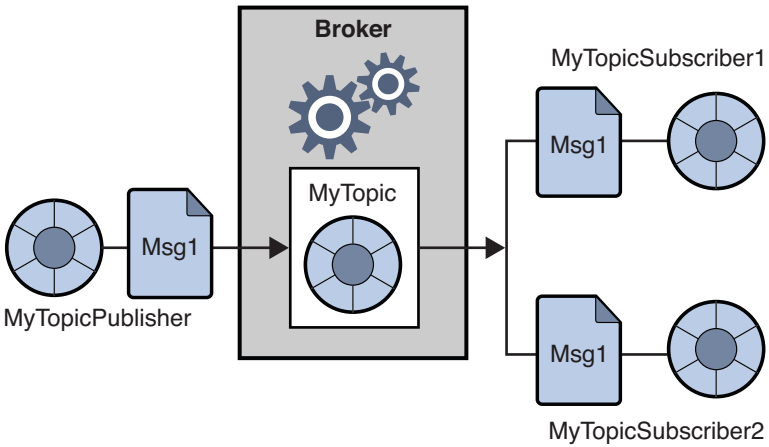

FIGURE 2–3 Simple Publish/Subscribe Messaging

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

Subscribers can be *durable* or *non-durable*. If a durable subscriber becomes inactive, the broker retains messages for it until the subscriber becomes active and consumes the messages. If a non-durable subscriber becomes ionactive, the broker does not retain messages for it.

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

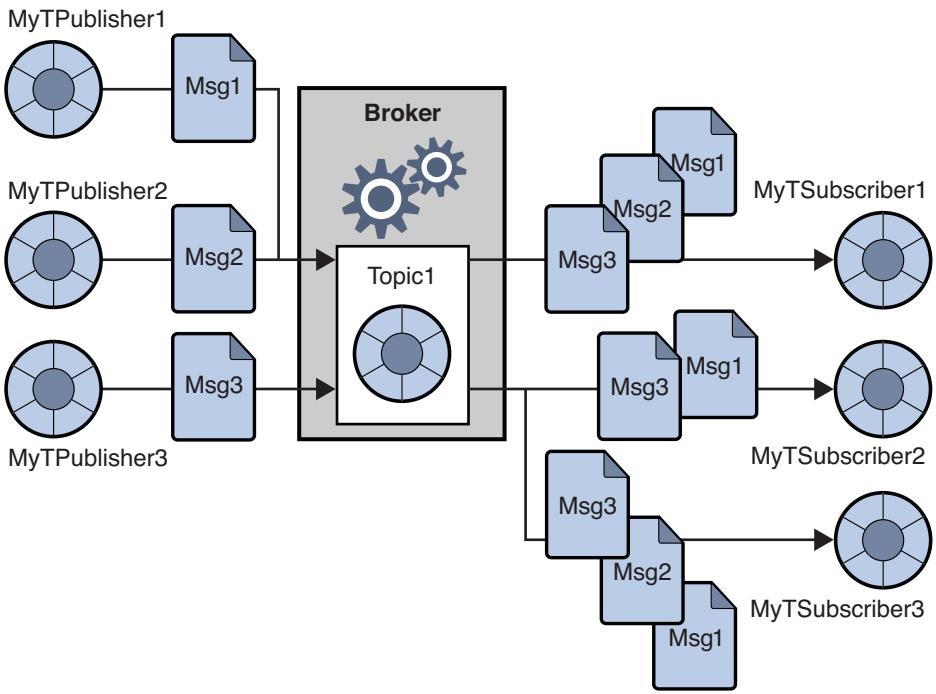

FIGURE 2–4 Complex Publish/Subscribe Messaging

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

- More than one publisher can publish messages to a topic. Publishers can share a connection or use different connections, but they can all access the same topic.
- More than one subscriber can consume messages from a topic. Subscribers consume all messages published to a topic unless they use selectors to filter out messages or the messages expire before they are consumed.

- Subscribers can share a connection or use different connections, but they can all access the same topic.
- For durable subscribers, the broker retains messages for the subscribers while these subscribers are inactive.
- Messages are placed in a topic in the order they are produced, but the order in which they are consumed depends on factors such as message expiration date, message priority, and whether a selector is used in consuming messages.
- Publishers and subscribers have a timing dependency: a topic subscriber can consume only messages published after the subscriber has subscribed to the topic.
- Publishers and subscribers can be added and deleted dynamically at runtime, thus allowing the messaging system to expand or contract as needed.

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

Domain-Specific and Unified APIs

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

TABLE 2–1 JMS Programming Domains and Objects

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

The unified domain was introduced with JMS version 1.1. The domain-specific API also provides a clean programming interface that prevents certain types of programming errors: for example, creating a durable subscriber for a queue destination. However, the domain-specific APIs have the disadvantage that you cannot combine point-to-point and publish/subscribe

operations in the same transaction or in the same session. If you need to do that, you should choose the unified domain API. See "The Request-Reply Pattern" on page 51 for an example of combining the two domains.

Programming Objects

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

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

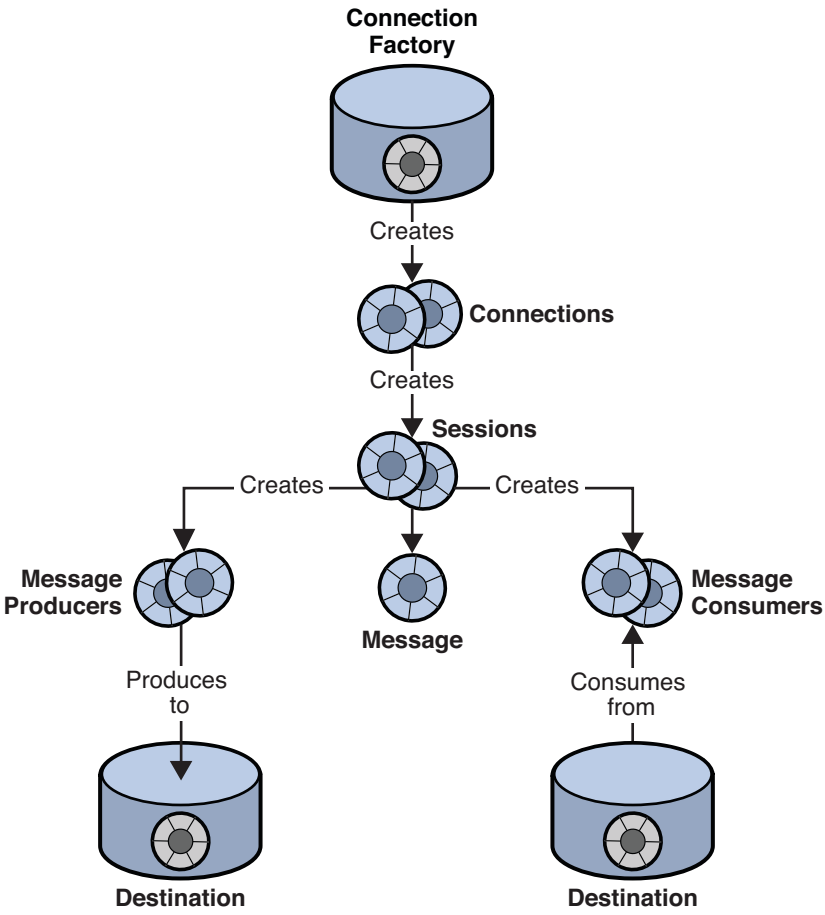

FIGURE 2–5 JMS Programming Objects

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

TABLE 2–2 Producing and Consuming Messages.

Producing a Message	Consuming a Message
1. The administrator creates a connection factory administered object.	
2. The administrator creates a physical destination and the administered object that corresponds to it.	
3. The client obtains a connection factory object through a JNDI lookup.	
4. The client obtains a destination object through a JNDI lookup.	

TABLE 2–2 Producing and Consuming Messages. *(Continued)*

Producing a Message	Consuming a Message
5. The client creates a connection and sets any properties that are specific to this connection.	
6. The client creates a session and sets the properties that govern messaging reliability.	
7. The client creates a message producer for a specified destination.	The client creates a message consumer for a specified destination.
8. The client creates a message.	The client starts the connection.
9. The client sends a message.	The client receives a message.

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

Connection Factories and Connections

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

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

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

- The name of the host on which the broker resides, the connection service desired, and the port through which the client is to access that service.
- How automatic reconnection to the broker should be handled if the connection fails. This feature reconnects the client to the same (or, in a broker cluster, to a different broker) if a connection is lost.
- The ID of any client that needs the broker to track its durable subscription.
- The default name and password of any user attempting the connection. This information is used to authenticate the user and authorize operations if a password is not specified at connection time.
- Whether broker acknowledgements should be suppressed for any clients that are not concerned with reliability.

- How to manage the flow of control and payload messages between the broker and the client runtime.
- How queue browsing should be handled (Java clients only).
- Whether certain message header fields should be overridden.

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

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

Sessions

If the connection represents a communication channel between the client and the broker, a session marks a single conversation between them. Mainly, you use a session object to create messages, message producers, and message consumers. When you create a session, you configure reliable delivery through a number of *acknowledgement* options or through transactions. For more information, see "Reliable Messaging" on page 53.

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

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

- Create and configure destinations for those clients that do not use administered objects to obtain references to existing destinations.
- Create and configure temporary topics and queues; these are used as part of the request-reply pattern. See "The Request-Reply Pattern" on page 51.
- Support transaction processing.
- Define a serial order for producing or consuming messages.
- Serialize the execution of message listeners for asynchronous consumers (see "Consuming a Message" on page 50).
- Create queue browsers (Java clients only).

Messages

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

Message Header

A header is required of every JMS message. The header contains ten predefined fields, which are listed and described in Table 2–3.

TABLE 2–3 JMS-Defined Message Header

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

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

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

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

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

Three of the header fields (`JMSDeliveryMode`, `JMSExpiration`, and `JMSPriority`) can be set at two different levels:

- For all messages produced by a specific message producer.
- For each message when it is produced.

If these fields are set at more than one level, values set when producing a message override those set for the message's producer.

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

Message Properties

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

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

Message Queue defines a number of additional message properties. These properties are used to identify compressed messages and how messages should be handled if they cannot be delivered. For more information see "Managing Message Size" in *Sun Java System Message Queue 4.1 Developer's Guide for Java Clients*.

Message Body

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

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

TABLE 2–4 Message Body Types

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

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

Producing a Message

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

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

The message producer can also implement a request-reply pattern by setting the `JMSReplyTo` message header field. For more information, see "The Request-Reply Pattern" on page 51.

Consuming a Message

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

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

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

These factors are described in the following sections.

Another factorthat affects message delivery, the degree of reliability required by the messaging application, is described in "Reliable Messaging" on page 53.

Synchronous and Asynchronous Consumers

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

- **Synchronous consumption** (pull model) means the consumer explicitly requests that a message be delivered and then consumes it.

  Depending on the method used to request messages, a synchronous consumer can choose to wait (indefinitely) until a message arrives, to wait a specified amount of time for a message, or to return immediately if there is no message ready to be consumed (messages that were successfully produced but which the broker has not finished processing).
- **Asynchronous consumption** (push model) means that the message is automatically delivered to a message listener object (`MessageListener`) that has been registered with the consumer. The client consumes the message when a session thread invokes the `onMessage()` method of the message listener object.

Using Selectors to Filter Messages

A message consumer can use a message selector to have the message service deliver only those messages whose properties (see "Message Properties" on page 48) match specific selection criteria. You specify this criteria when you create the consumer.

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

```
color = ”red’
size > 10
```

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

Using Durable Subscribers

A durable subscriber is one for which the broker retains messageseven when the subscriber becomes inactive.

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

The Request-Reply Pattern

You can combine producers and consumers in the same connection (or even session when using the unified API). In addition, the JMS API allows you to implement a request-reply pattern for your messaging operations by using *temporary destinations*. Temporary destinations are explicitly created and destroyed programmatically. They are maintained by the broker only for the duration of the connection in which they are created.

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

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

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

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

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

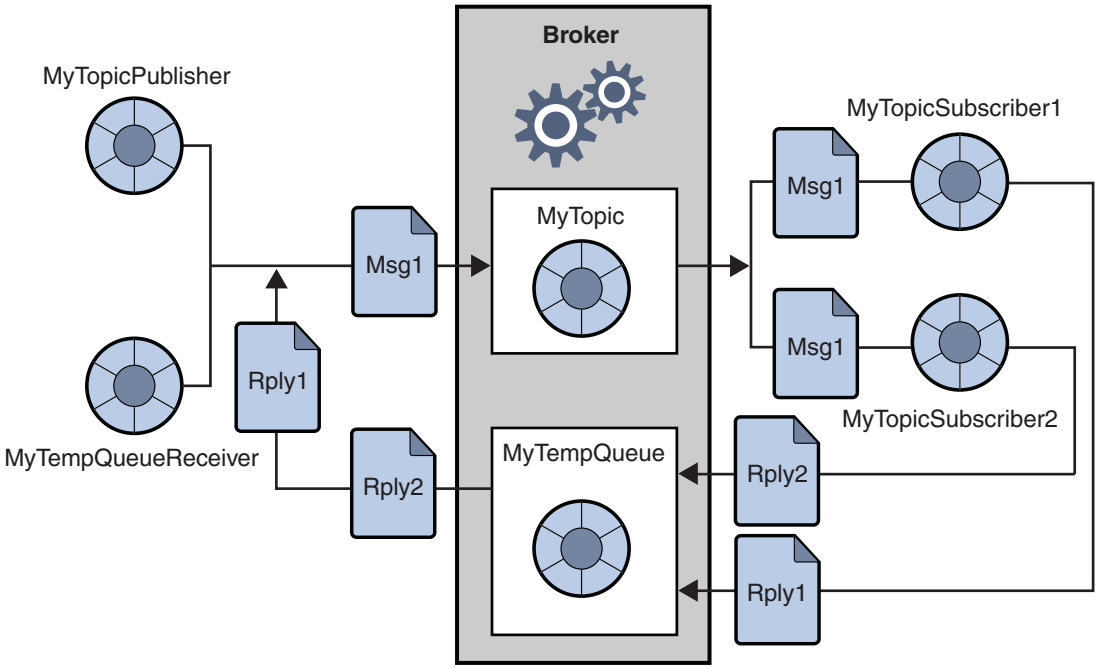

FIGURE 2–6 Request/Reply Pattern

As the figure shows, `MyTopicPublisher` produces `Msg1` to the destination `MyTopic`. `MyTopicSubsriber1` and `MyTopicSubscriber2` consume the message and send a reply to `MyTempQueue`, from which `MyTempQueueReceiver` retrieves it. This pattern might be useful for an application that publishes pricing information to a large number of subscribers and which queues their (reply) orders for sequential processing.

Temporary destinations last only as long as the connection in which they are created. While any producer can produce messages to a temporary destination, the only consumers that can access a temporary destination are those created in the same connection in which the temporary destination was created.

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

- If you anticipate that the connection in which the temporary destination was created might terminate before the reply is sent.
- If reply messages need to be persistent.

Reliable Messaging

Message delivery occurs in two hops: the first hop takes the message from the producer to a physical destination on the broker; the second hop takes the message from that destination to the consumer. Thus, a message can be lost in one of three ways:on its hop from the producer to the broker, on its hop from the broker to the consumer, and while it's in broker memory (if the broker fails). Reliable delivery guarantees that delivery will not fail in any of these ways.

Two mechanisms are used to ensure reliable delivery:

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

The following sections describe these two aspects of ensuring reliability.

Note – Reliable delivery only applies to messages for which the `JMSDeliveryMode` message header field indicates a persistent message.

Acknowledgements

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

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

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

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

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

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

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

Transactions

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

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

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

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

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

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

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

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

Persistent Storage

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

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

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

Chapter 3, "The Message Queue Broker," describes the default message store supplied by the Message Queue service and how an administrator can set up and configure an alternate store.

A Message's Journey Through the System

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

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

Message delivery is illustrated in Figure 2–7.

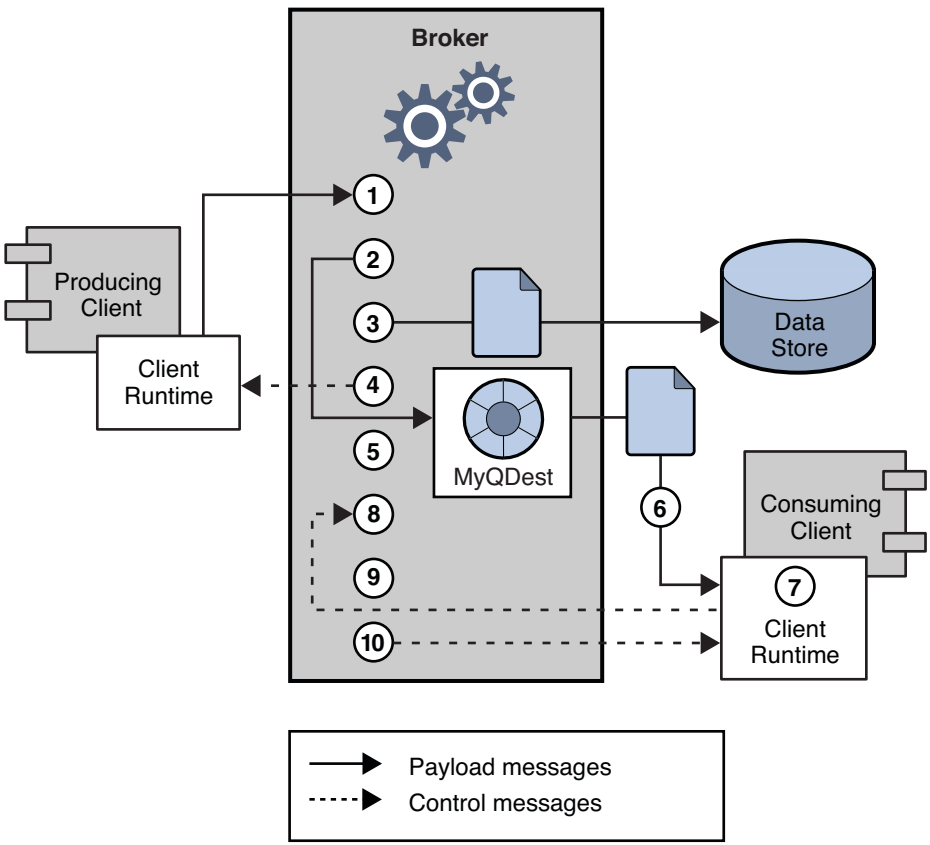

FIGURE 2–7 Message Delivery Steps

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

Message Production

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

Message Handling and Routing

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

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

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

5. The broker determines the routing for the message.

6. The broker writes out the message from its destination to the appropriate connection, tagging it with a unique identifier for the consumer.

Message Consumption

7. The message consumer's client runtime delivers the message from the connection to the message consumer.

8. The message consumer's client runtime acknowledges consumption of the message to the broker.

Message End-of-Life

9. The broker processes the client acknowledgement, and deletes the (persistent) message when all acknowledgements have been received.

10. The broker confirms to the consumer's client runtime that the client acknowledgement has been processed.

The broker can discard a message before it is consumed if the administrator deletes the message from a destination or if the administrator removes or redefines a durable subscriber, thereby causing a message in a topic destination to be removed without it being delivered. The broker can also discard a message before it is consumed if the message has expired, if memory limits

have been reached, or if delivery fails due to a client exception. If you don't want a message discarded in these situations, you can have the broker store the messages in a special destination called the *dead message queue*. Storing messages in the dead message queue allows you to troubleshoot the system and recover messages in any of these situations.

Design and Performance

The behavior of a Message Queue application depends on many factors: client design, connection configuration, broker configuration, broker tuning, and resource management. Some of these are the responsibility of the application developer; others are the concern of the Message Queue administrator.

In the best of possible worlds the developer should be aware of how the Message Queue service can support and scale the application design, and the administrator should be aware of the application's design goals when it comes time to tune the application. Messaging behavior can be optimized through redesign as well as through careful monitoring and tuning. Thus, a key aspect of creating a good Message Queue application is for the developer and the administrator to understand what can be realized at each stage of the application life cycle and to share information about desired and observed behavior.

Chapter 3, "The Message Queue Broker," explains how you can use the Message Queue service to support, manage, and tune messaging performance.

Working with SOAP Messages

Simple Object Access Protocol (SOAP) allows for the exchange of structured data (specified by an XML schema) between two peers in a distributed environment. Sun's implementation of SOAP does not currently support reliable SOAP messaging nor does it support publishing SOAP messages. However, you can use the Message Queue service to achieve reliable SOAP messaging and, if desired, to publish SOAP messages. The Message Queue service does not deliver SOAP messages directly, but it allows you to wrap SOAP messages into JMS messages, to produce and consume these messages like normal JMS messages, and to extract the SOAP message from the JMS message.

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

To achieve reliable SOAP messaging you need to implement the following sequence of actions:

1. Use the Message Transformer utility to convert the SOAP message into a JMS message.

2. Send the JMS message to the desired destination.
3. Consume the JMS message asynchronously or synchronously.
4. After the JMS message is consumed, use the Message Transformer utility to convert it into a SOAP message.
5. Use the SOAP with Attachments API for Java (SAAJ) API (defined in the `java.xml.soap` package) to disassemble the SOAP message.

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

Java and C Clients

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

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

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

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

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

The C interface, does not support the following features:

- The use of administered objects
- Map, stream, or object message types
- Consumer-based flow control
- Queue browsers
- JMS application server facilities (Connection Consumer, distributed transactions)
- Receiving or sending SOAP messages
- Receiving or sending compressed JMS messages

- Auto-reconnect or failover, which allows the client runtime to automatically reconnect to a broker if a connection fails
- The `NO_ACKNOWLEDGE` mode

◆ ◆ ◆ CHAPTER 3

The Message Queue Broker

This chapter provides a more detailed view of the Message Queue broker, which was introduced in "The Message Queue Service" on page 27. The chapter examines the services provided by the broker, the tools you use to configure these services, and the administrative tasks required to support these services.

The chapter includes the following sections:

Broker Services

Figure 1–6 shows the different elements of the Message Queue service. Chapter 2, "Client Programming Model," described the programming model and how clients use the Java and C APIs to interact with the Message Queue client runtime, the part of the message service that is directly accessed by client applications. This chapter focuses on the broker services, the part of the message service that is accessed through administration tools.

The broker is the centerpiece of the Message Queue service shown in Figure 1–6. The broker provides the set of services that enable secure, reliable messaging:

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

- **Monitoring services** that generate metrics and diagnostic information and write this information to a specified output channel.

These broker services are configured by setting broker properties. The sections that follow describe each of the services and summarize the broker properties that you can use to customize that service to meet the needs of your messaging application.

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

Connection Services

You use connection-related properties to configure and manage the physical connections between a broker and its clients. Connection services available for Message Queue clients are introduced in "Connection Services" on page 31, which describes available connection services: their name, type, and underlying protocol.

Connection services are multithreaded and available through dedicated ports that can be dynamically assigned by the broker's *Port Mapper* (see "Port Mapper Service" on page 63) or statically assigned by the administrator. By default, when you start the broker, the `jms` and `admin` services are up and running. Additionally, you can configure a broker to run any or all of the connection services.

Connection configuration can be performed by both administrators and in client application code:

- An administrator creates connection factory administered objects that encapsulate connection behaviors. In addition, an administrator can use broker properties to activate non-default connection services, to assign static ports if required, to configure threading, and to specify a host to connect to if multiple network cards are used. An administrator can also specify a ping interval to test whether the client is accessible; this is useful in managing resources.
- Client code can instantiate configuration factory objects and set their attributes to achieve desired connection behaviors. These attributes specify non-default connection services, hosts, ports, a list of brokers to connect to in case reconnection is required, and reconnection behavior. The client can also specify a ping interval to test for failed connections.

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

Each connection service also supports specific authentication and authorization features. See "Security Services" on page 67 for more information.

Port Mapper Service

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

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

Thread Pool Management

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

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

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

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

Routing and Delivery Services

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

Physical Destinations

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

- **Admin-created destinations** are created by an administrator using Message Queue administration tools. Admin-created destinations correspond to destination administered objects created by an administrator and accessed by client applications by using a JNDI lookup. Admin-created destinations can also correspond to destination objects created programmatically by a client application. You use Message Queue administration tools to set or update properties for each admin-created destination.
- **Auto-created destinations** are automatically created by the broker whenever a message consumer or producer attempts to access a nonexistent destination. These are typically used during development. You can set a broker property to disallow the creation of such destinations. You set broker properties to configure all auto-create destinations on a particular broker.

  An auto-created destination is automatically destroyed by the broker when it is no longer being used: that is, when it has no consumer clients and no longer contains any messages. If a broker restarts, it only recreates this kind of destination if it contains persistent messages.
- **Temporary destinations** are explicitly created and destroyed programmatically by client applications that need a destination at which to receive replies to messages. As their name implies, these destinations are temporary. They are maintained by the broker only for the duration of the connection in which they are created.

  Temporary destinations are not stored persistently and are never recreated when a broker is restarted, but they are visible to administration tools.
- **The dead message queue** is a specialized destination, created automatically at broker startup and used to store dead messages for diagnostic purposes. You can set properties for the dead message queue using the `imqcmd` utility.

Managing Destinations

Managing a destination involves one or more of the following tasks:

- Creating, pausing, resuming, or destroying a destination
- Listing all destinations on a broker
- Displaying information about the state and properties of a destination

- Displaying metrics information for a destination
- Compacting disk space used to persist messages for a destination
- Updating a physical destination's properties

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

Configuring Physical Destinations

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

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

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

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

Managing Memory

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

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

Destination Message Limits

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

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

System-Wide Message Limits

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

System Memory Thresholds

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

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

Persistence Services

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

The Message Queue service supports both file-based and JDBC compliant persistence modules (see Figure 3–1). File-based persistence is the default.

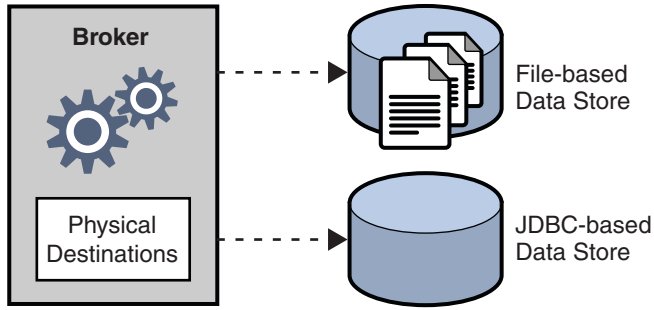

FIGURE 3–1 Persistence Support

File-Based Persistence

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

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

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

JDBC-Based Persistence

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

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

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

Security Services

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

- *Authentication* ensures that only verified users can establish a connection to the broker.

- *Authorization* specifies which users or groups have the right to access resources and to perform specific operations.
- *Encryption* protects messages from being tampered with during delivery over a connection.

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

Figure 3–2 shows the components needed by the broker to provide authentication and authorization.

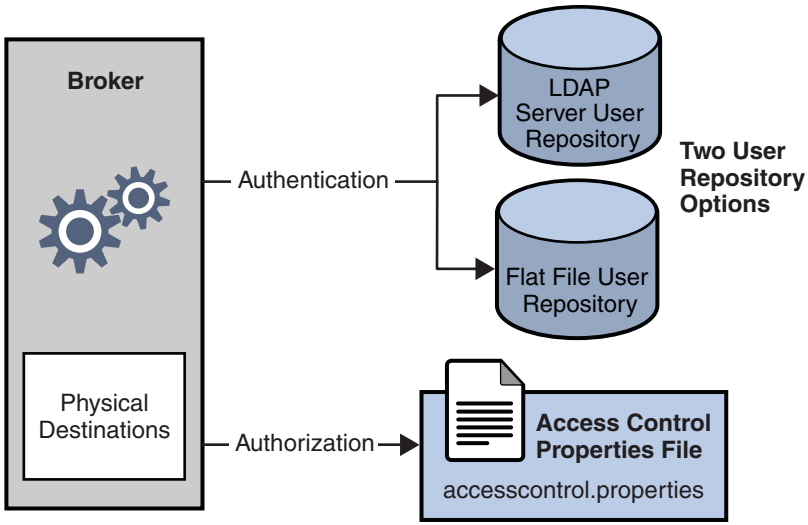

FIGURE 3–2 Security Manager Support

As Figure 3–2 shows, you can store user data in a flat file user repository that is provided with the Message Queue service or you can plug in a pre-existing LDAP repository. You set a broker property to indicate your choice.

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

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

Authentication and Authorization

When a client requests a connection, the client must supply a user name and password. The broker compares the specified name and password to those stored in the user repository. On transmitting the password from client to broker, the passwords are encoded using either base 64 encoding or message digest (MD5) hashing. MD5 is used for a flat file repository; base 64 is required for LDAP repositories. If using LDAP you may want to use the secure TLS protocol. You can set broker properties to configure the type of encoding used by each connection service separately or to set the encoding on a broker-wide basis.

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

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

You set broker properties to specify the following information:

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

JAAS-Based Authentication

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

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

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

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

Encryption

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

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

Monitoring Services

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

- 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 metrics message producer that sends JMS messages containing metrics information to topic destinations for consumption by JMS monitoring clients.
- An MBean server that implements the Java Management Extensions (JMX) API
- Support for the Java ES Monitoring Framework

The following subsections describe these components..

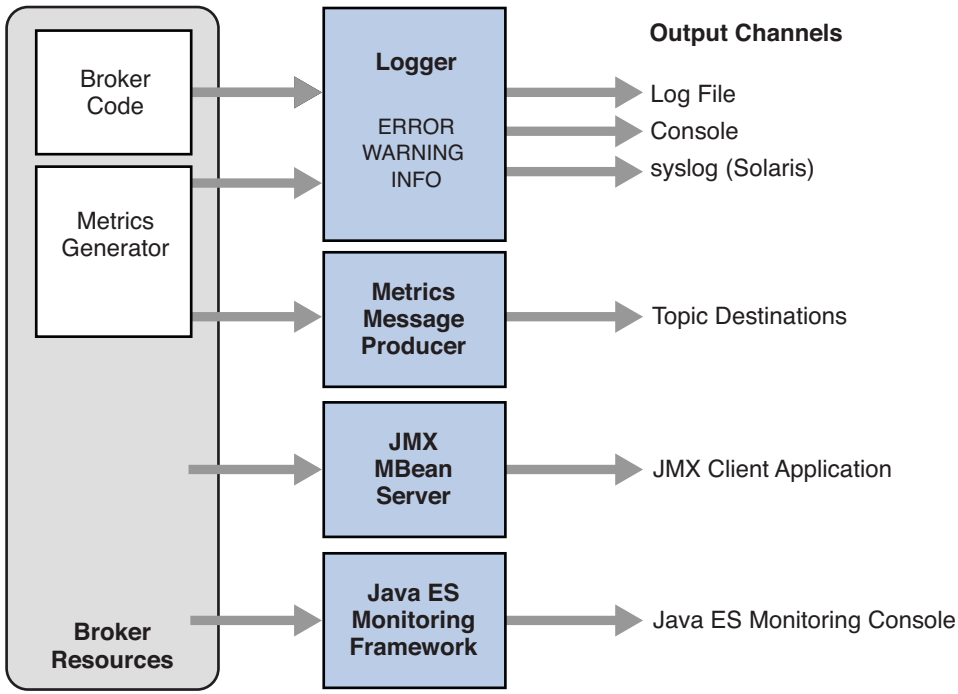

FIGURE 3–3 Monitoring Service Support

Metrics Generator

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

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

Logger

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

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

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

Metrics Message Producer

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

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

JMX MBean Server

The broker's MBean Server implements the Java Management Extensions (JMX) API. Using this API, you can perform broker configuration and monitoring operations programmatically from within a Java application.

Using the JMX API, a Java application can access data values representing static or dynamic properties of a broker, connection, destination, or other resource. The application can also receive notifications of state changes or other significant events affecting the resource.

For more information see "JMX-Based Administration" on page 75.

Java ES Monitoring Framework Support

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

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

- the installed product

- the broker instance name
- the broker port mapper
- each connection service
- each physical destination
- the persistent store
- the user repository

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

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

Administration Tools

This section describes the tools you use to configure Message Queue broker services.

Built-in Administration Tools

The following illustration showsthe part of the Message Queue service that represents the administration tools provided by Message Queue forconfiguring and managing broker services.

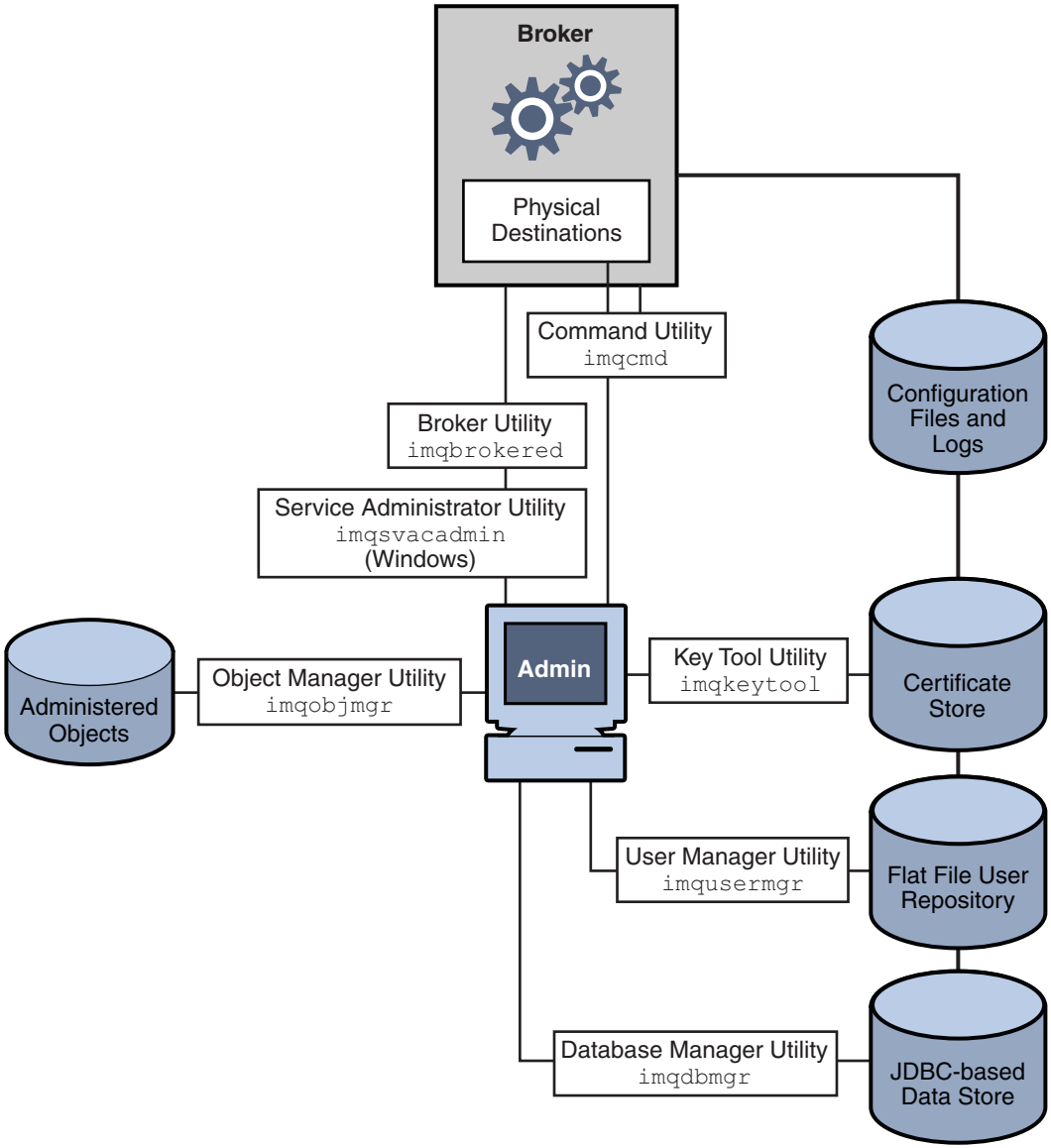

FIGURE 3–4 Command-Line Administration Tools

The administration tools include the following command line interfaces:

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

- **Command utility** (`imqcmd`). Used after starting a broker to manage broker resources, such as connection services, connections, durable subscriptions, transactions, physical destinations, and so forth.
- **Object Manager utility** (`imqobjmgr`). Used to create, list, update, and delete administered objects in a JNDI object store.
- **User Manager utility** (`imqusermgr`). Used to populate a file-based user repository for user authentication and authorization.
- **Database Manager utility**(`imqdbmgr`). Used to create and manage a JDBC-based persistent data store. (The built-in file store requires no external management.)
- **Key Tool utility** (`imqkeytool`). Used to generate self-signed broker certificates needed for SSL authentication.
- **Service Admiistrator utility** (`imqsvcadmin`).Used to install, query, and remove a broker as a Windows service.

Message Queue administration tools also include a GUI-based administration interface. The Administration Console combines some of the capabilities of the Command utility (`imqcmd`) and the Object Manager utility (`imqobjmgr`). You can use it to do the following:

- Manage a broker, its connection services, and other resources.
- Create, update, and delete physical destinations.
- Connect to a JNDI object store, add administered objects to the store, and manage them.

JMX-Based Administration

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

- *Resources* include everything that you can manipulate using the Command utility (`imqcmd`) and the Message Queue Admin Console: the broker, communication services, connections, destinations, durable subscribers, transactions, and so on.
- *Management* includes the ability to dynamically configure and monitor resources, and the ability to obtain notifications about state changes and error conditions.

The JMX specification defines an architecture for the instrumentation and programmatic management of distributed resources. This architecture is based on the notion of a *managed bean,* or *MBean:* a Java object, similar to a JavaBean, representing a resource to be managed. Message Queue MBeans may be associated with individual resources such as brokers, connections, or destinations, or with whole categories of resources, such as the set of all destinations on a broker. There are separate *configuration MBeans* and *monitor MBeans* for setting a resource's configuration properties and monitoring its runtime state.

Java applications obtain MBeans through an *MBean server,* which serves as a container and registry for MBeans. Each Message Queue broker contains an MBean server, accessed by means of a *JMX connector.* The JMX connector is used to obtain an *MBean server connection,* which in turn provides access to individual MBeans on the server.

The JMX specification defines an architecture that enables the programmatic management of any distributed resource. This architecture is defined by design patterns, APIs, and various services.

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

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

JMX is the Java standard for building management applications and is widely used for managing Java EE infrastructure. If your Message Queue client is a part of a larger Java EE deployment, JMX support allows you to use a standard programmatic management framework throughout your Java EE application. Message Queue is based on the JMX 1.2 specification, which is part of JDK 1.5.

To manage a Message Queue broker using the JMX architecture, see *Sun Java System Message Queue 4.1 Developer's Guide for JMX Clients*.

Administration Tasks

This section describes the tasks that you need to complete to support a Message Queue development or a production environment.

Supporting a Development Environment

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

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

Supporting a Production Environment

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

Setup Operations

Typically, you have to perform the following setup operations:

- Secure administrative access

  Whether you use a file-based or LDAP user repository, make sure that the administrator is in the `admin` group and has a secure password. If necessary, create a secure connection to the broker for the administrator.

- Secure client access

  Whether you use a file-based or LDAP user repository, populate the user repository with the names of users who can access the message service and edit the access control properties file to give them appropriate authorization. If necessary set up SSL-based connection services. To prevent unauthenticated connections, be sure to change the "guest" user's password.

- Create and configure physical destinations

  Set destination attributes so that the number of messages and the amount of memory allocated for messages can be supported by broker resources.

- Create and configure administered objects.

  If you want to use an LDAP object store, configure and set up the store. Create and configure connection factory and destination administered objects.

- If horizontal scaling and/or message service availability is required, create a broker cluster.

  For a conventional broker cluster, create a cluster configuration file and designate a master broker.

For a high-availability (HA) broker cluster, create a cluster configuration file that specifies HA property values.

Maintenance Operations

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

- Support and manage application clients
  - Monitor and manage destinations, durable subscriptions, and transactions
  - Disable auto-create capability
  - Monitor and manage the dead message queue
- Monitor and tune the broker
  - Recover failed brokers
  - Monitor, tune, and reconfigure the broker
  - Manage broker memory resources
  - Expand clusters if necessary
- Manage administered objects

  Create additional administered objects as needed and adjust connection factory attributes to improve performance and throughput.

◆ ◆ ◆ CHAPTER 4

Broker Clusters

Message Queue supports the use of *broker clusters*: groups of brokers working together to provide message delivery services to clients. Clusters enable an Message Queue service to scale messaging operations by distributing client connections among multiple brokers. Clusters also provide for message service availabillty by using multiple brokers to protect against individual broker failure.

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

Cluster Models

Message Queue supports two clustering models both of which provide a scalable message service, but with each providing a different degree of availability:

- **Conventional broker clusters.** A conventional broker cluster provides *message service availability*. When a broker or a connection fails, clients connected to the failed broker reconnect to another broker in the cluster. However, messages and state information stored in the failed broker cannot be recovered until the failed broker is brought back online. This can result in an interruption of message delivery.
- **High-availability broker clusters.** A high-availability broker cluster provides *data availability* in addition to message service availability. When a broker or a connection fails, another broker takes over the pending work of the failed broker. The failover broker has access to the failed broker's messages and state information. Clients connected to the failed

broker reconnect to the failover broker. In a high-availability cluster, as compared to a conventional cluster, a failure results in no interruption of message delivery.

Conventional and high-availability broker clusters are built on the same underlying infrastructure and message delivery mechanisms. They differ in how brokers in the cluster are synchronized with one another and in how the cluster detects and responds to failures.

The sections that follow first describe the infrastructure and delivery mechanisms common to both clustering models, after which the unique aspects of each model is explained.

Cluster Message Delivery

A broker cluster facilitates the delivery of messages between client applications that are connected to different brokers in the cluster.

The following illustration shows salient features of a Message Queue broker cluster. Each of three brokers is connected to the other brokers: in the cluster: the cluster is fully-connected. The brokers communicate with each other and pass messages by way of a special *cluster connection service*, shown in Figure 4–1 by the dashed lines.

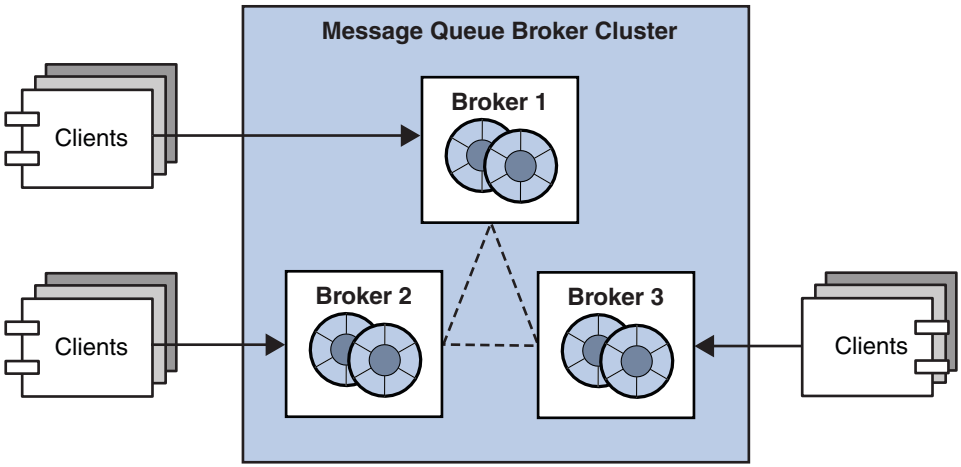

FIGURE 4–1 Message Queue Broker Cluster

Each broker typically has a set of messaging clients (producers and/or consumers) that are directly connected to that broker. For these client applications, the broker to which they are directly connected is called their home broker. Each client communicates directly only with its home broker, sending and receiving messages as if that broker were the only broker in the cluster.

Accordingly, a producer in the cluster produces messages to a destination in its home broker. The home broker is responsible for routing and delivering the messages to all consumers of the destination, whether these consumers be local (connected to the home broker) or remote (connected to other brokers in the cluster). The home broker works in concert with the other brokers to deliver messages to all connected consumers.

To facilitate delivery of messages across the cluster, information about the destinations and consumers of each broker is propagated to all brokers in the cluster. Each broker therefore stores the following information:

- The name, type, and properties of all physical destinations in the cluster
- The name, location, and destination interests of each message consumer

Changes in this information are propagated whenever one of the following events occurs:

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

The propagation of destination and consumer information across the cluster means that destinations and consumers are essentially global to the cluster. In the case of destinations, most properties set for a physical destination (see **Broken Link (Target ID: AERCN)**) apply to all instances of that destination in the cluster. However a few properties apply to each individual destination instance (the maximum number of messages, the maximum number of message bytes, and the maximum number of producers).

Despite the global nature of destinations and consumers in a cluster, a home broker has special responsibilities with respect to both its producers and consumers:

- A producer's home broker is responsible for persisting and routing messages originating from that producer, for logging, for managing transactions, and for processing acknowledgements from consuming clients across the cluster.
- A consumer's home broker is responsible for persisting information about consumers, for forwarding remotely produced messages to the consumer, for letting a producer's home broker know whether the consumer is still available, and for letting a producer's home broker know when each message has been successfully consumed.

The cluster connection service passes messages from destinations on a home broker to destinations on remote brokers, and passes control messages such as client acknowledgements from remote brokers back to a home broker. The cluster attempts to minimize message traffic across the cluster. For example, it only sends a message to a remote broker if there is a remote consumer of the message, and if a remote broker has two identical subscribers for the same

topic destination, the message is sent over the wire only once. (You can further reduce traffic by setting a destination property specifying that delivery to local consumers has priority over delivery to remote consumers.)

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

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

Conventional Clusters

The following figure illustrates a conventional broker cluster.

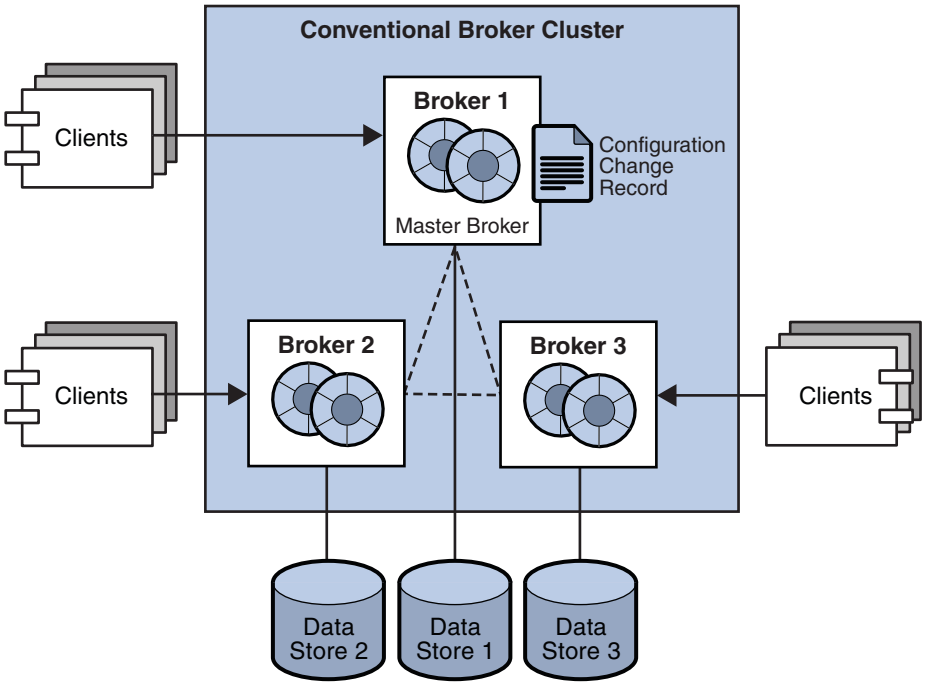

FIGURE 4–2 Conventional Broker Cluster

A conventional broker cluster has the following characteristics:

- **Data Synchronization**

Each broker has its own respective persistent data store in which destinations, persistent messages, and other state information is stored. Some of this information (fore example, destinations and durable subscriptions) has been propagated to the broker from other brokers in the cluster. If a broker fails, it is possible for this information to become out of sync with the information stored by other brokers in the cluster. To guard against this possibility in a conventional broker, one broker within the cluster is designated as the *master broker*. The master broker maintains a *configuration change record* to track changes to the cluster's propagated persistent entities.

When an offline broker comes back online (or when a new broker is added to the cluster), it consults the configuration change record for information about destinations and durable subscribers, then exchanges information with other brokers about currently active message consumers.

Because brokers cannot complete their initialization without accessing the configuration change record, the master broker should always be the first broker started within the cluster. Furthermore, if the master broker goes offline, information cannot be propagated across the cluster. Under these conditions, you will get an exception if you try to create, reconfigure, or destroy a destination or a durable subscription or attempt a related operation. (Non-administrative message delivery continues to work normally, however.)

- **Failure Detection**

  A conventional broker cluster has no internal mechanism for detecting the failure of a broker or of the connection between brokers. Failure is detected when clients are unable to interact with their home broker. Otherwise, it is the responsibility of an administrator to monitor brokers in the cluster using Message Queueadministration tools (see "Administration Tools" on page 73.

- **Client Reconnect**

  If a broker or its connection to a client fails, the client automatically reconnects to another broker in the cluster. This broker becomes the client's new home broker. The reconnect is governed by connection properties that specify the order and frequency by which the client attempts to reconnect to brokers in the cluster.

  However, in this scenario, the new home broker does not have all the client-related state information that was previously held by the failed broker. For example, messages to have been consumed by the client or the state of transactions involving the client might have been lost. As a result, the failure of a broker in a conventional cluster can cause interruptions in message delivery.

High-Availability Clusters

The following figure illustrates a high-availability (HA) broker cluster. A high-availability broker provides both service availability and data availability.

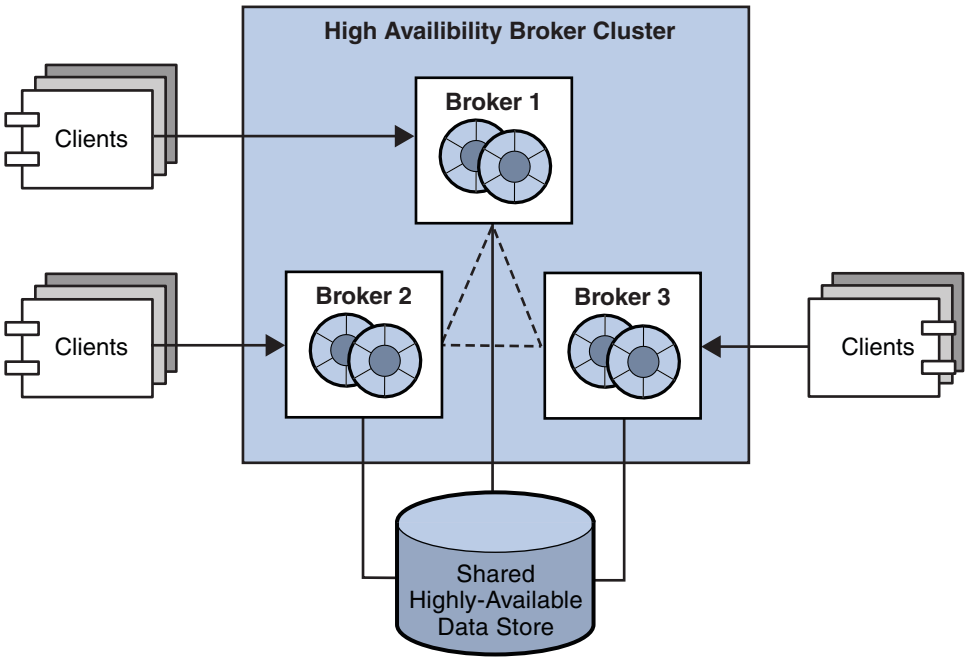

FIGURE 4–3 High Availability Cluster

A high-availability broker cluster has the following characteristics:

- **Data Synchronization**

  All brokers in a high-availability cluster share a common persistent data store in which destinations, persistent messages, and other state information is stored for each broker. To achieve data availability, the shared data store is a highly available JDBC database. Because all brokers share the same data store, each broker is able to access the state information stored by other brokers in the cluster. When a broker that has been offline rejoins the cluster (or when a new broker is added to the cluster) it is able to access the most current information simply by accessing the shared data store. Similarly, if a broker fails, another broker is able to access and take over the failed broker's information in the shared data store.

- **Failure Detection**

A high-availability cluster makes use of a distributed heartbeat service by which brokers inform other brokers that they are online and accessible by the cluster connection service. The heartbeat service also updates broker state information in the cluster's shared data store. When no heartbeat packet is detected from a broker for a configurable number of heartbeat intervals, the broker is considered suspect of failure. The other brokers in the cluster then begin to monitor the suspect broker's state information in the shared data store to confirm whether the broker has indeed failed. If the suspect broker fails to update its state information within a configurable interval, it is considered to have failed. There is a trade-off between the speed and accuracy of failure detection: configuring the cluster for quick failure detection increases the likelihood that a slow broker will be considered to have failed.

If these services operating in tandem determine that a broker has failed, then a failover broker is selected from among the remaining online brokers to take over the pending work of the failed broker.

When one broker detects that another broker has failed, it will attempt to take over the failed broker's persistent state (pending messages, destinations, durable subscriptions, pending acknowledgments, and open transactions) so as to provide uninterrupted service to the failed broker's clients. If two or more brokers attempt such a takeover, only the first will succeed (the first acquires a lock on the failed broker's data in the persistent store, preventing subsequent takeover attempts). After an initial waiting period (for clients to reconnect to the failover broker), the failover broker will clean up any transient resources (such as transactions and temporary destinations) belonging to the failed broker.

- **Client Reconnect**

  If a broker fails, its clients reconnect to the failover broker, which becomes their new home broker. The reconnect is governed by connection properties that specify the order and frequency by which clients attempt to reconnect to brokers in the cluster. In this case, if a client attempts to reconnect to a broker that is not the failover broker, the reconnect is rejected and the client is redirected to the failover broker.

  In this scenario, the new home broker (the failover broker) has immediate access to all the client-related state information that was previously held by the failed broker and can therefore take over where the failed broker left off. As a result, the failure of a broker in a high-availablity cluster will not cause interruptions in message delivery.

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

Cluster Models Compared

Conventional and high-availability cluster models share the same basic infrastructure. They both use the cluster communication service to enable message delivery between producers and consumers across the cluster. However, as shown in the following figure and described in previous sections, these models differ in how destination and consumer information is synchronized across the cluster, in the mechanisms for detecting failure, in how client reconnect takes place.

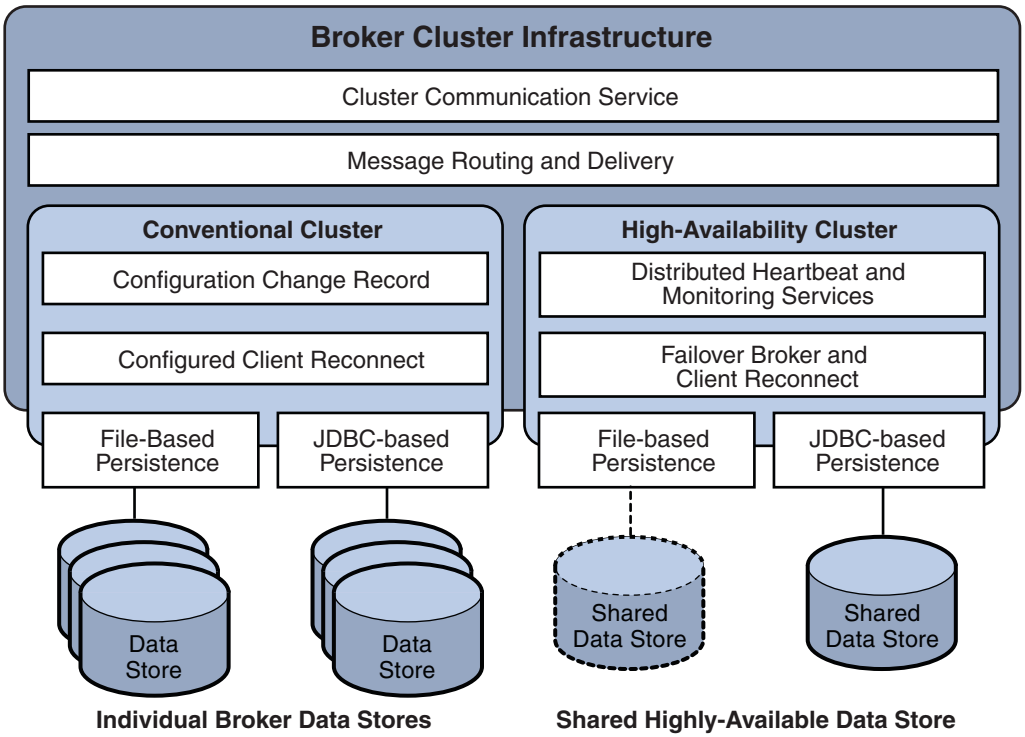

FIGURE 4–4 Cluster Infrastructure

In addition, while both models rely on the broker's persistent data store interfaces (both flat-file and JDBC), in the case of high-availability clusters the shared data store must be a highly available JDBC database.

The following table summarizes the functional differences between the two cluster models. This information might help in deciding which model to use or whether to switch from one to another.

TABLE 4–1 Clustering Model Differences

Functionality	Conventional	High-Availability
Performance	Faster than high-availability model.	Slower than conventional model
Service availability	Yes, but some operations are not possible if master broker is down.	Yes.
Data availability	No. State information in failed broker is not available.	Yes at all times.
Transparent recovery from failure	No. Message delivery is interrupted. Also, client reconnects might not be possible if failure occurs during a transaction commit (rare).	Yes. No interruption in message delivery. Client reconnects might not be possible if failure occurs during a transaction commit (extremely rare).
Configuration	Set appropriate cluster configuration properties for each broker.	Set appropriate cluster configuration properties for each broker.
Additional requirements	None.	Highly available database.
Remote sites supported	Yes.	No.

Cluster Configuration

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

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

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

For complete information about administering broker clusters, see Chapter 8, "Broker Clusters," in *Sun Java System Message Queue 4.1 Administration Guide*. For information about the effect of reconnection on the client, see "Connection Event Notification" in *Sun Java System*

Message Queue 4.1 Developer's Guide for Java Clients and "Client Connection Failover (Auto-Reconnect)" in *Sun Java System Message Queue 4.1 Developer's Guide for Java Clients*.

Information to Be Deleted

Note – The information in the next two sections ("Destination Attributes" and "Clustering and Destinations") is being discarded as being redundant, or too detailed for this book, or too confusing, or too inaccurate, or some or all of these.

Destination Attributes

Attributes set for a physical destination on a clustered broker apply to all instances of that destination in the cluster; however, some limits specified by these attributes apply to the cluster as a whole and others to individual destination instances. This behavior is the same for both clustering models. Table 4–2 lists the attributes you can set for a physical destination and specifies their scope.

TABLE 4–2 Properties for Physical Destinations on Clustered Brokers

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

Clustering and Destinations

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

Producing to a Queue Using the Reply-To Model

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

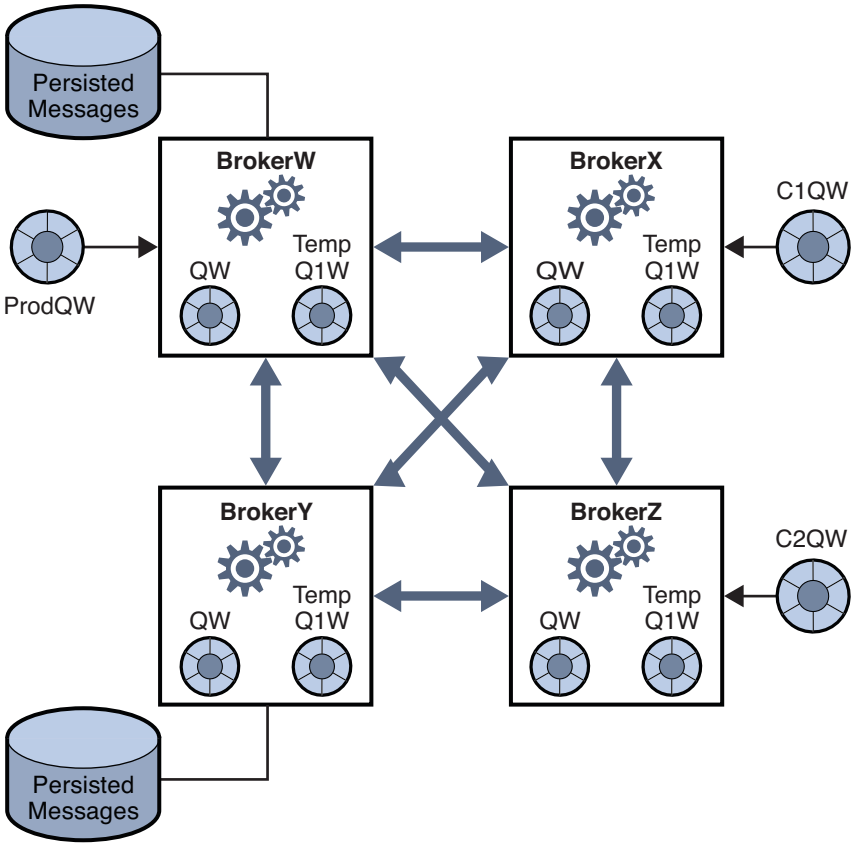

FIGURE 4–5 Replication of Destinations in a Cluster: Queue with Reply-To

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

Producing to an Auto-Created Destination

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

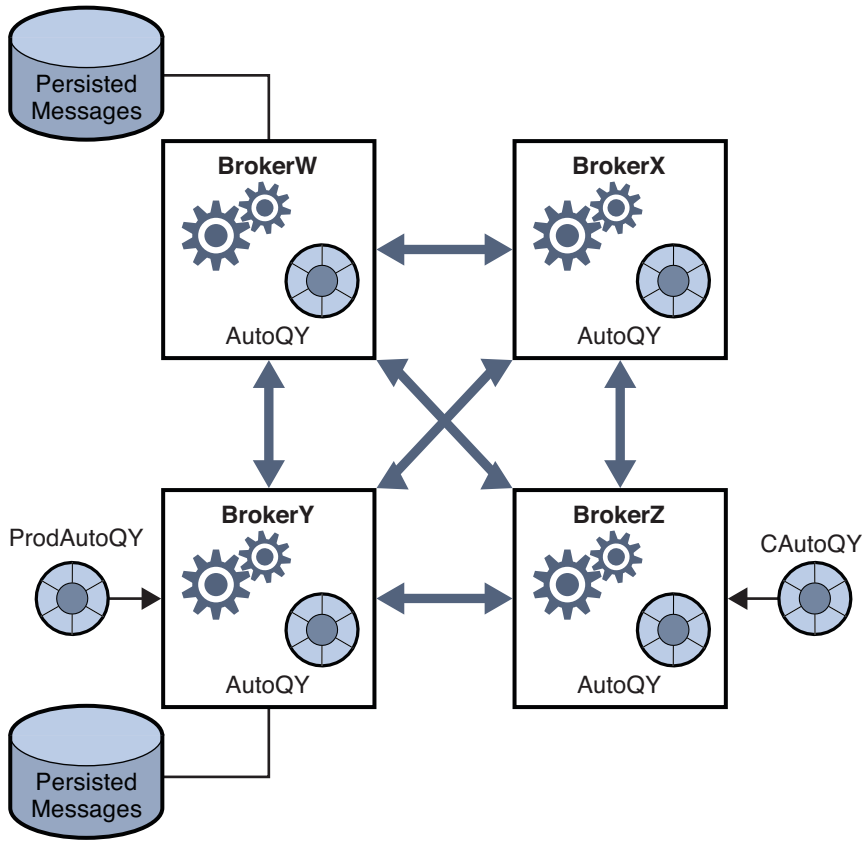

FIGURE 4–6 Replication of Destinations in a Cluster: Auto-Created Destinations

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

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

Publishing to a Topic Destination

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

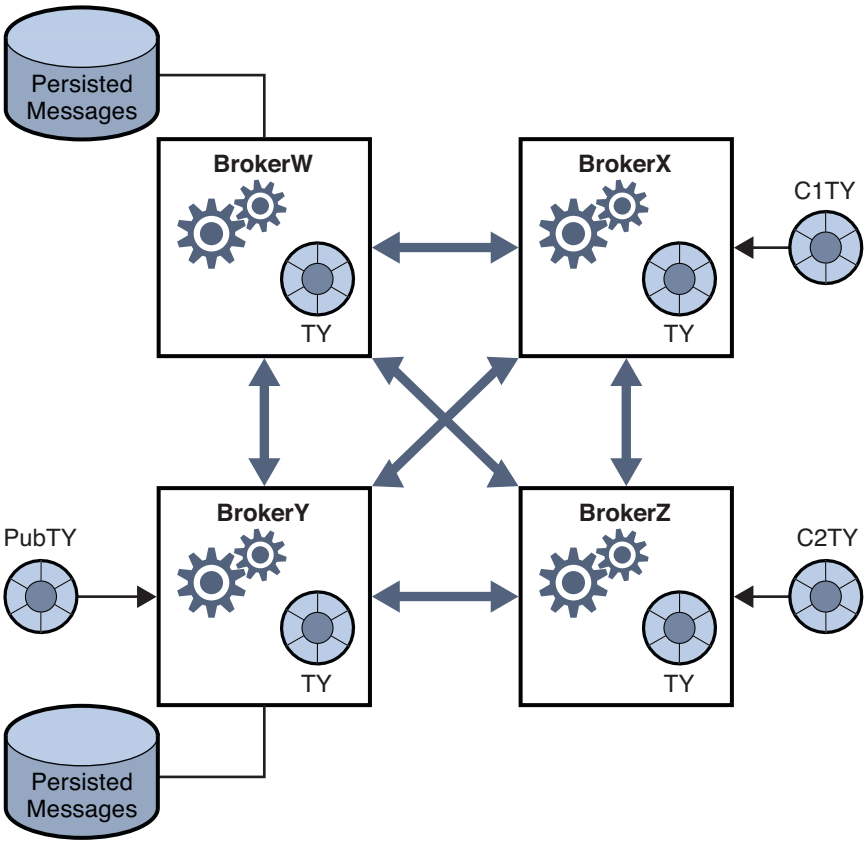

FIGURE 4–7 Replication of Destinations in a Cluster: Publishing to a Topic

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

Handling Destinations in the Event of Connection or Broker Failure

Table 4–3 explains how different kinds of destinations are replicated and deleted in a cluster.

TABLE 4–3 Handling Destinations in a Cluster

Destination	Propagation, and Deletion
Admin-created	When the destination is created it is propagated in the cluster, and each broker stores information about the destination persistently. The destination is destroyed when the administrator explicitly deletes it. Using the conventional cluster model, if there is a master broker, a record of the creation and deletion is stored in the master broker to allow brokers in the cluster to synchronize state information. Using the high availability cluster model, information synchronized using the shared persistent store.
Temporary	When the destination is created, it is propagated around the cluster. If the consumer associated with the temporary destination is allowed to reconnect, the destination is persistently stored on the consumer's home broker. Otherwise, the destination is never stored. In this case, if the consumer loses its connection, the destination is deleted on all brokers. If the consumer's home broker crashes and the consumer is allowed to reconnect, temporary destinations associated with this consumer are monitored. If the consuming client does not reconnect within a specific period of time, it is assumed that the client has failed and the destination is deleted.
Auto-created	When a producer is created and a destination does not exist, the destination is created on the producer's home broker. When a consumer is created for a destination that does not exist, information about the consumer and the destination is propagated across the cluster. An auto-created destination can be explicitly deleted by an administrator, or it can be automatically deleted ■ By each broker when there have been no consumers or messages for a given period of time. ■ By each broker, when the broker restarts and there are no messages for that destination.

◆ ◆ ◆ CHAPTER 5

Message Queue and Java EE

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

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

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

JMS/Java EE Programming: Message-Driven Beans

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

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

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

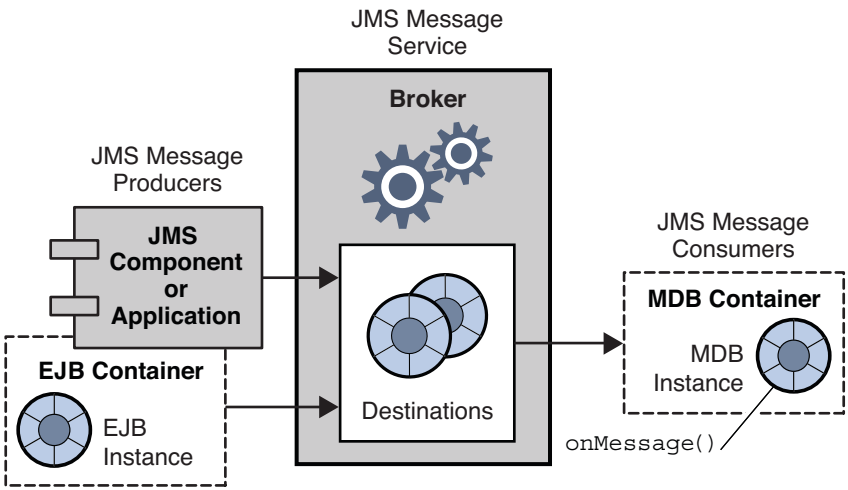

FIGURE 5–1 Messaging with MDBs

- A JMS message driven bean is an EJB that implements the 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 5–1.
- A specialized EJB container supports the MDB. It creates instances of the MDB and sets them up for asynchronous consumption of messages. The container sets up a connection with the message service (including authentication), creates a pool of sessions associated with a given destination, and manages the distribution of messages among the pooled sessions. Since the container controls the life cycle of MDB instances, it manages the pool of MDB instances to accommodate incoming message loads.

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

Java EE Application Server Support

In Java EE architecture, EJB containers are hosted by Java EE 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 Java EE 1.4 Application Server, Message Queue is plugged into the application server as an embedded JMS resource adapter.

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

For information about Java EE architecture, see the Java EE Platform Specification located at `http://java.sun.com/javaee/downloads/index.jsp`.

JMS Resource Adapter

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

Plugging a JMS resource adapter into an application server allows Java EE 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 Java EE application server administration tools.

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

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

The Message Queue resource adapter is a single file (`imqjmsra.rar`) located in a directory that depends on the operating system (see Chapter 17, "JMS Resource Adapter Property Reference,"

in *Sun Java System Message Queue 4.1 Administration Guide*). The `imqjmsra.rar` file contains the resource adapter deployment descriptor (`ra.xml`) as well as the JAR files needed by the application server in order to use the adapter.

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

◆ ◆ ◆ 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.

Table A–1 describes how the Message Queue service handles JMS optional items.

Optional Features

TABLE A–1 Optional JMS Functionality

Section in JMS Specification	Description and Message Queue Implementation
3.4.3 `JMSMessageID`	"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." **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 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." **Message Queue implementation**: The Message Queue product supports administrative override of the values in message header fields through configuration of the client runtime (see "Message Header" on page 47).

TABLE A–1 Optional JMS Functionality *(Continued)*

Section in JMS Specification	Description and Message Queue Implementation
3.5.9 JMS Defined Properties	"JMS Reserves the '`JMSX`' Property name prefix for JMS defined properties.""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 Appendix B, "Stability of Message Queue Interfaces," in *Sun Java System Message Queue 4.1 Administration Guide*).
3.5.10 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.
4.4.8 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 54).
4.4.9 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 "Point-To-Point Messaging" on page 38.

◆ ◆ ◆ APPENDIX B

Message Queue Features

The Message Queue service fully implements the JMS 1.1 specification for reliable, asynchronous message delivery. For information about JMS compliance-related issues, see Appendix A, "Message Queue Implementation of Optional JMS Functionality."

Message Queue has additional capabilities and features that exceed JMS requirements. You can use these features to integrate and monitor systems consisting of large numbers of distributed components exchanging many thousands of messages in round-the-clock, mission-critical operations.

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

Feature List

TABLE B–1 Message Queue Features

Feature	Description and Reference
Administration tools	The Message Queue service includes GUI and command line tools for managing destinations, transactions, durable subscriptions, administered object stores, user repositories, JDBC-compliant data stores, and server certificates. **Reference** "Built-in Administration Tools" on page 73. Chapter 1, "Administrative Tasks and Tools," in *Sun Java System Message Queue 4.1 Administration Guide*

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Authentication	Authenticate users seeking a connection to the broker. The Message Queue service allows users to connect to the broker by validating their name and password against values stored in a user repository. The repository can be a flat-file repository shipped with Message Queue or an LDAP repository (LDAP v2 or v3 protocol). **To Use** 1. Create a user repository or use the default instance. 2. Use the `imqusermgr` tool to populate the repository. **JAAS-Based Authentication** Application clients can also use authentication services based on the Java Authentication and Authorization Service (JAAS), which allows you to plug in a variety of services into the broker to authenticate Message Queue clients. The JAAS API is a core API in J2SE and therefore it is an integral part of Message Queue's runtime environment. **To Use** 1. The JAAS provider supplies a login module class that implements the authentication service. 2. Obtain JAAS configuration file and specify its location using a system property. 3. Configure broker properties that relate to JAAS support. **Reference** "Authentication and Authorization" on page 69. Chapter 9, "Security," in *Sun Java System Message Queue 4.1 Administration Guide*.

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Authorization	Authorize users to perform specific operations. The Message Queue service allows you to create an access control properties file that specifies the operations users and groups of users can perform. The broker checks this file when a client seeks to create a connection, create a producer, create a consumer, or browse a queue. **To Use** Edit the access control properties file that is automatically created for the broker instance. **Reference** "Authentication and Authorization" on page 69 Chapter 9, "Security," in *Sun Java System Message Queue 4.1 Administration Guide*.
Automatic reconnect	The administrator sets connection attributes on the connection factory administered object to enable automatic reconnection in the event of connection or broker failure. Reconnection can be to the same broker or to another broker in a cluster if a cluster is used. You can specify how many times to try reconnection and the interval between attempts. You can also specify how often to iterate through a list of brokers and whether to iterate through the list in a specific order. **Reference** "Connection Services" on page 62. Chapter 7, "Administered Objects," in *Sun Java System Message Queue 4.1 Administration Guide*.

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Broker clusters	The administrator can balance client connections and message delivery across a number of broker instances by grouping those instances into a broker cluster. The Message Queue service supports two kinds of clusters: conventional clusters and high availability clusters **To Use Conventional Clusters** 1. Specify cluster configuration properties for each broker in the cluster. Specify properties that are the same for all brokers using a cluster configuration file. 2. If there is a master broker, start the master broker 3. Start the other brokers in the cluster. **To Use High Availability Clusters** 1. Specify cluster configuration properties for each broker in the cluster (including JDBC-related properties). Specify properties that are the same for all brokers using a cluster configuration file. 2. Install your JDBC driver's `.jar` file in the appropriate directory location. 3. Use the `imqdbmgr` tool to create the database schema for the highly available data store. 4. Start the brokers in the cluster. **Reference** Chapter 4, "Broker Clusters" Chapter 8, "Broker Clusters," in *Sun Java System Message Queue 4.1 Administration Guide*.
Broker configuration	The administrator can set broker properties to tune Message Queue service performance. This includes routing services, persistence services, security, monitoring, and administered object management. **Reference** Chapter 3, "The Message Queue Broker" Chapter 4, "Broker Configuration," in *Sun Java System Message Queue 4.1 Administration Guide*

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
C client support	C clients can use Message Queue messaging services to send and receive messages. The C API enables legacy C applications and C++ applications to participate in JMS-based messaging. 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. **Reference** "Java and C Clients" on page 59. *Sun Java System Message Queue 4.1 Developer's Guide for C Clients*
C-API support for distributed transactions	The Message Queue C-API supports the XA interface (between a distributed transaction manager and Message Queue as a XA-compliant resource manager), allowing Message Queue C-API clients running in a distributed transaction processing environment (such as BEA Tuxedo) to participate in distributed transactions. **Reference**
Client runtime logging	Java clients can use all the J2SE 1.4 logging facilities to configure how the Message Queue client runtime outputs its logging information. Clients can choose to log the following events: changes in connection state and miscellaneous connection activities, session-related events, the creation of producers, consumers, and destinations, and the consumption and production of messages. Java clients can configure logging programmatically or by using configuration files. **Reference** "Client Runtime Logging" in *Sun Java System Message Queue 4.1 Developer's Guide for Java Clients*

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Compressed messages	Java clients can set a message property to have the client runtime compress a message being sent. The runtime on the consumer side decompresses the message before it delivers it to the consumer. Additional properties are provided that you can use to determine whether compressing messages would actually improve performance. **Reference** "Message Body" on page 48. "Managing Message Size" in *Sun Java System Message Queue 4.1 Developer's Guide for Java Clients*.
Configurable persistence	The administrator can configure the broker to use the file-based persistent store provided with Message Queue or a JDBC-compliant database, such as Oracle 8i. **To Use** Set broker properties that relate to file-system persistent storage or JDBC-compliant storage. **Reference** "Persistence Services" on page 66. "Configuring a Persistent Data Store" in *Sun Java System Message Queue 4.1 Administration Guide*.

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Configurable physical destinations	The administrator can define some messaging behavior by setting physical destination properties when creating destinations. The following behavior can be configured for any destination: the maximum number of unconsumed messages or the maximum amount of memory allowed for such messages, which messages the broker should reject when memory limits are reached, the maximum number of producers and consumers, the maximum message size, the maximum number of messages delivered in a single batch, whether the destination can deliver only to local consumers, and whether dead messages on the destination can be moved to the dead message queue. **Reference** "Routing and Delivery Services" on page 64. Chapter 6, "Physical Destinations," in *Sun Java System Message Queue 4.1 Administration Guide*.
Connection event notification	Java clients can listen for connection events (like closure or reconnection) and take appropriate action based on the notification type and the connection state. **To Use** 1. Use the event notification API to create an event listener. 2. Add code to the client application that will take appropriate action depending on the events captured by the event listener. **Reference** "Connection Event Notification" in *Sun Java System Message Queue 4.1 Developer's Guide for Java Clients*

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Connection ping	The administrator can set a connection factory attribute to specify the frequency of a ping operation from the client runtime to the broker. This allows the client to preemptively detect a failed connection. **Reference** "Connection Services" on page 62. "Connection Services" in *Sun Java System Message Queue 4.1 Administration Guide*.
Dead message queue	The Message Queue message service creates the dead message queue to hold messages that have expired or that the broker could not process. You can examine the contents of the queue to monitor, tune, or troubleshoot system performance. **Reference** "Routing and Delivery Services" on page 64. Chapter 6, "Physical Destinations," in *Sun Java System Message Queue 4.1 Administration Guide*.

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
HTTP connections	Java clients can create HTTP connections to the broker. HTTP transport allows messages to be delivered through firewalls. Message Queue implements HTTP support using an HTTP tunnel servlet that runs in a web server environment. Messages produced by a client are wrapped by the client runtime as HTTP requests and delivered over HTTP through a firewall to the tunnel servlet. The tunnel servlet extracts the JMS message from the HTTP request and delivers the message over TCP/IP to the broker. **To Use** 1. Deploy HTTP tunnel servlet on a web server. 2. Configure broker's `httpjms` connection service and start the broker. 3. Configure HTTP connection. 4. Obtain an HTTP connection to the broker. (Java clients only.) **Reference** **Broken Link (Target ID: AERAW)**. Appendix C, "HTTP/HTTPS Support," in *Sun Java System Message Queue 4.1 Administration Guide*
Interactive monitoring	The administrator can use the `imqcmd metrics` command to monitor a broker remotely. Monitored data includes JVM metrics, broker message flow, connections, connection resources, messages, destination message flow, destination consumers, destination resource use. **Reference** "Monitoring Services" on page 70. Chapter 10, "Monitoring Broker Operations," in *Sun Java System Message Queue 4.1 Administration Guide*

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Java EE resource adapters	Message Queue provides a resource adapter that can be plugged into a Java EE-compliant application server. By using Message Queue as a JMS provider, an application server meets the Java EE requirement that distributed components running in the application server be able to interact using reliable, asynchronous message. **To Use** Configure the adapter by setting adapter attributes. **Reference** "Java EE Application Server Support" on page 97. Chapter 17, "JMS Resource Adapter Property Reference," in *Sun Java System Message Queue 4.1 Administration Guide*.
Java ES Monitoring Framework support	The Java ES Monitoring Framework allows administrators to use the same interface to manage any and all Java ES components. If you are using Message Queue with other Java ES components, it might be more convenient to manage these from a single console. Administrators can use the Sun Java System Monitoring Console to view performance statistics, create rules to monitor automatically, and acknowledge alarms. To enable Java ES monitoring, you must do the following: ■ Install and configure the components in your deployment; for example, Message Queue and the Application Server. ■ Enable and configure the Monitoring Framework for all your monitored components. ■ Install the Monitoring Console on a separate host, start the master agent, and then start the web server. For information, see the *Sun Java Enterprise System Monitoring Guide*.

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
JMX-Based Administration	Java clients can use the JMX API to monitor and manage broker resources: the broker, services, connections, destinations, consumers, producers, and so on. You can use JMX-based management in different ways to monitor application performance, to monitor the broker, to automate tasks, or to write custom tools. **Reference** *Sun Java System Message Queue 4.1 Developer's Guide for JMX Clients*
JNDI service provider support	Clients can look up administered objects using the JNDI API. Administrators can use the `imqobjmgr` utility to add, list, update, and delete administered objects in an object store accessible using JNDI. **Reference** "Built-in Administration Tools" on page 73 Chapter 7, "Administered Objects," in *Sun Java System Message Queue 4.1 Administration Guide*
LDAP Server support	An administrator can use an LDAP server as a Message Queue administered object store and as a user repository (needed for authentication). By default Message Queue provides file-based storage for this data. **To Use as an Administered Object Store** 1. Use the tools provided by the LDAP vendor to set up the LDAP server. 2. Set the LDAP-related broker properties to define the initial context and the location of the object store. 3. Set the LDAP-related broker properties that relate to securing the LDAP server operations. **Reference** Chapter 9, "Security," in *Sun Java System Message Queue 4.1 Administration Guide*

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Memory resource management	The administrator can configure the following behavior: 1. Set properties on a destination to specify the maximum number of producers, the maximum number of messages, and the maximum size of any one message. 2. Set properties on a destination to control message flow. 3. Set properties on a destination to manage message flow for each destination. 4. Set properties on the broker to specify message limits on all destinations for that broker. 5. Set properties on the broker to specify thresholds of available system memory at which the broker takes action to prevent memory overload. The action taken depends on the state of memory resources. **Reference** "Routing and Delivery Services" on page 64. Chapter 4, "Broker Configuration," in *Sun Java System Message Queue 4.1 Administration Guide*
Message compression	The developer can set a message header property to have the client runtime compress a message before sending it. The client runtime on the consumer side decompresses the message before delivering it to the consumer. **Reference** "Message Properties" on page 48. "Message Compression" in *Sun Java System Message Queue 4.1 Developer's Guide for Java Clients*

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Message flow control to clients	The administrator or the developer can configure a connection to specify various flow limits and metering schemes to minimize the collision of payload and control messages, and thereby to maximize message throughput. **To Use** Set the flow-control attributes for the connection factory administered object (administrator), or set the flow-control properties for the connection factory (developer). **Reference** "Connection Factories and Connections" on page 45. "Connection Services" in *Sun Java System Message Queue 4.1 Administration Guide*. "Connection Factory Attributes" in *Sun Java System Message Queue 4.1 Administration Guide*
Message-based monitoring API	Java clients can use a monitoring API to create custom monitoring applications. A monitoring application is a consumer that retrieves metrics messages from special metrics topic destinations. **To Use** 1. Write a metrics monitoring client. 2. Set broker properties to configure the broker's metrics message producer. 3. Set access controls on metrics topic destinations. 4. Start the monitoring client. **Reference** "Monitoring Services" on page 70. Chapter 4, "Using the Metrics Monitoring API," in *Sun Java System Message Queue 4.1 Developer's Guide for Java Clients*. Chapter 10, "Monitoring Broker Operations," in *Sun Java System Message Queue 4.1 Administration Guide*.

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Multiple destinations for publishers and subscribers	Publishers can publish messages to multiple topic destinations and subscribers can consume messages from multiple topic destinations by using a destination name that includes wildcard characters, representing multiple destinations. Using such symbolic names allows administrators to create additional topic destinations, as needed, consistent with the wildcard naming scheme. Publishers and subscribers automatically publish to and consume from the added destinations. (Wildcard destination subscribers are more common than publishers.) **Reference**
Queue delivery to multiple consumers	Clients can register more than one consumer for a given queue. The administrator can specify the maximum number of active consumers and the maximum number of backup consumers for the queue. The broker distributes messages to the registered consumers, balancing the load among them in order to allow the system to scale. **To Use** Set physical destination properties `maxNumActiveConsumers` and `maxNumBackupConsumers`. **Reference** "Point-To-Point Messaging" on page 38. Chapter 15, "Physical Destination Property Reference," in *Sun Java System Message Queue 4.1 Administration Guide*.

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Reliable data persistence	To obtain absolute reliability you can require that the operating system write the data synchronously to the persistent store by setting the `imq.persist.file.sync.enabled` property to true. This eliminates possible data loss due to system crashes, but at the expense of performance. Note that although the data is not lost, it is not available to any other broker (in a cluster) because data is not currently shared by clustered brokers. When the system comes back up, the broker can reliably resume operations. **Reference** "Persistence Services" on page 66. "Persistence Properties" in *Sun Java System Message Queue 4.1 Administration Guide* .
Schema validation of XML messages	Enables validation of the content of a text (not object) XML message against an XML schema at the point the message is sent to the broker. The location of the XML schema (XSD) is specified as a property of a Message Queue destination. If no XSD location is specified, the DTD declaration within the XML document is used to perform DTD validation. (XSD validation, which includes data type and value range validation, is more rigorous than DTD validation.)

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Secure connections	Clients can secure transmission of messages using the Secure Socket Layer (SSL) standard over TCP/IP and HTTP transports. These SSL-based connection services allow for the encryption of messages sent between clients and broker. 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. **To Use** 1. Generate a self-signed or signed certificate. 2. Enable the secure service. 3. Start the broker. 4. Configure client security connection properties and run the client. **Reference** **Broken Link (Target ID: AERAW)**. "Security Services" on page 67. Chapter 9, "Security," in *Sun Java System Message Queue 4.1 Administration Guide*. "Working With Secure Connections" in *Sun Java System Message Queue 4.1 Developer's Guide for C Clients*

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Simple Object Access Protocol (SOAP) support	Clients can receive SOAP (XML) messages and they can wrap them as JMS messages and use Message Queue to exchange them as they would a JMS message. Clients can use a special servlet to receive SOAP messages; they can use a utility class to wrap a SOAP message as a JMS message; they can use another utility class to extract the SOAP message from the JMS message. Clients can use standard SOAP with Attachments API for Java (SAAJ) libraries to assemble and disassemble a SOAP message. **Reference** "Working with SOAP Messages" on page 58. Chapter 5, "Working with SOAP Messages," in *Sun Java System Message Queue 4.1 Developer's Guide for Java Clients*.
Support for Sun Connection Registration	The Message Queue installer allows for registration of Message Queue with Sun Connection, a Sun-hosted service that helps you track, organize, and maintain Sun hardware and software. **Reference**
Thread management	The administrator can specify the maximum and minimum number of threads assigned to any specific connection service. The administrator can also determine whether a connection service could increase throughput by using a shared thread model, which allows threads dedicated to idle connections to be used by other connections. **To Use** Set connection service thread-related properties. **Reference** "Thread Pool Management" on page 63. Chapter 4, "Broker Configuration," in *Sun Java System Message Queue 4.1 Administration Guide*.

TABLE B–1 Message Queue Features *(Continued)*

Feature	Description and Reference
Tunable performance	The administrator can set broker properties to adjust memory usage, threading resources, message flow, connection services, reliability parameters, and other elements that affect message throughput and system performance. **Reference** "Monitoring Services" on page 70. "Monitoring Services" in *Sun Java System Message Queue 4.1 Administration Guide* Chapter 11, "Analyzing and Tuning a Message Service," in *Sun Java System Message Queue 4.1 Administration Guide*

Glossary

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

acknowledgement Control messages exchanged between clients and broker 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 broker.

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 broker on behalf of the client.

client runtime Message Queue software that provides messaging clients with an interface to the Message Queue message service. The client runtime supports all operations needed for clients to send messages to destinations and to receive messages from destinations.

cluster Two or more interconnected brokers that work in concert to provide scalable messaging services. In the event of failover and reconnection, conventional clusters provide service availability; high availability clusters provide service and data availability.

cluster connection service A private protocol that enables brokers in a cluster to provide reliable, synchronized service.

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

connection factory The administered object the client uses to create a connection to a broker. 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 broker to which produced messages are delivered for routing and subsequent delivery to consumers. This physical destination is identified and encapsulated by an administered object that a client uses to specify the destination for which it is producing messages and/or from which it is consuming messages.

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 service A middleware service that provides asynchronous, reliable exchange of messages between distributed components or applications. It includes a broker, the client runtime, the several data stores needed by the broker to carry out its functions, and the administrative tools needed to configure and monitor the broker and to tune performance.

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

L

M

O

P

Q

R

S

T

U

X