Administrator's Guide

Sun™ ONE Message Queue

Version 3.0.1

817-0354-10
October, 2002

Contents

List of Figures

List of Tables

List of Procedures

Preface

This book, the Sun™ ONE Message Queue (MQ) 3.0.1 *Administrator's Guide*, provides the background and information needed to perform administration tasks for an MQ messaging system.

This preface contains the following sections:

- Audience for This Guide
- Organization of This Guide
- Conventions
- Other Documentation Resources

Audience for This Guide

This guide is meant for administrators as well as application developers who need to perform MQ administration tasks.

An MQ administrator is responsible for setting up and managing an MQ messaging system, in particular the MQ message server at the heart of this system. The book does not assume any knowledge or understanding of messaging systems.

The guide is also meant to be used by application developers to better understand how to optimize their applications to make best use of the features and flexibility of the MQ messaging system.

Organization of This Guide

This guide is designed to be read from beginning to end. The following table briefly describes the contents of each chapter:

Table 1 Book Contents

Chapter	Description
Chapter 1, "Overview"	Presents a high-level conceptual overview of MQ messaging systems and terminology.
Chapter 2, "The MQ Messaging System"	Describes the MQ messaging system, with special emphasis on the MQ broker and the MQ client runtime that together provide messaging services.
Chapter 3, "MQ Administration"	Describes MQ administration tasks and tools, and introduces the command line utilities used for administration, and their common features.
Chapter 4, "Administration Console Tutorial"	Provides a hands-on tutorial to acquaint you with the Administration Console, a graphical interface to the MQ message server.
Chapter 5, "Starting and Configuring a Broker"	Explains how to start up and configure an MQ broker and a broker cluster.
Chapter 6, "Broker and Application Management"	Explains how to perform (application-independent) tasks related to managing MQ brokers, as well as tasks used to manage messaging applications.
Chapter 7, "Managing Administered Objects"	Explains how to perform tasks related to creating and managing MQ administered objects.
Chapter 8, "Security Management"	Explains how to perform security tasks related to applications, such as managing authentication, authorization, and encryption.
Appendix A, "Setting Up Plugged-in Persistence"	Explains how to set up MQ to use JDBC-compliant database to perform persistence functions.
Appendix B, "HTTP/HTTPS Support (Enterprise Edition)"	Explains how to set up HTTP connection services between a messaging client and the MQ message server.
Appendix C, "Using a Broker as a Windows Service"	Explains how to use the MQ Service Administration utility (`imqsvcadmin`) to install, query, and remove the broker (running as an Windows service).
Appendix D, "Location of MQ Data"	Describes the location of various categories of MQ data.

Table 1 Book Contents *(Continued)*

Chapter	Description
Appendix E, "Stability of MQ Interfaces"	Describes the stability of various MQ interfaces.
"Glossary"	Defines terms used in MQ documentation.

Conventions

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

Text Conventions

Table 2 Document Conventions

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

Directory Variable Conventions

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

Table 3 MQ Directory Variables

Variable	Description
`IMQ_HOME`	This is generally used in MQ documentation to refer to the root MQ installation directory: • On Solaris, there is no root MQ installation directory. Therefore, `IMQ_HOME` is not used in MQ documentation to refer to file locations on Solaris. • On Solaris, for Sun ONE Application Server, Evaluation Edition, the root MQ installation directory is: *root Application Server installation directory*`/imq`. • On Windows, `the` root MQ installation directory is set by the MQ installer (by default, as `C:\Program Files\Sun Microsystems\Message Queue 3.0.1`). • On Windows, for Sun ONE Application Server, the root MQ installation directory is: *root Application Server installation directory*`/imq`. • On Linux, the root MQ installation directory is, by default: `/opt/imq`.
`IMQ_VARHOME`	This is the `/var` directory in which MQ 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 ONE Application Server, Evaluation Edition, `IMQ_VARHOME` defaults to `IMQ_HOME/var`. • On Windows `IMQ_VARHOME` defaults to `IMQ_HOME/var`. • On Windows, for Sun ONE Application Server, `IMQ_VARHOME` defaults to `IMQ_HOME/var`. • On Linux, `IMQ_VARHOME` defaults to `IMQ_HOME/var`.

Table 3 MQ Directory Variables *(Continued)*

Variable	Description
IMQ_JAVAHOME	This is an environment variable that points to the location of the Java runtime (JRE 1.4) required by MQ executables: • On Solaris, IMQ_JAVAHOME defaults to the /usr/j2se/jre directory, but a user can optionally set the value to wherever JRE 1.4 resides. • On Windows, IMQ_JAVAHOME defaults to IMQ_HOME/jre, but a user can optionally set the value to wherever JRE 1.4 resides. • On Linux, IMQ_JAVAHOME defaults to the /usr/java/j2sdk1.0/jre directory, but a user can optionally set the value to wherever JRE 1.4 resides.

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

Other Documentation Resources

In addition to this guide, MQ provides additional documentation resources.

The MQ Documentation Set

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

Table 4 MQ Documentation Set

Document	Audience	Description
MQ *Installation Guide*	Developers and administrators	Explains how to install MQ software on Solaris, Linux, and Windows platforms.
Release Notes	Developers and administrators	Includes descriptions of new features, limitations, and known bugs, as well as technical notes.
MQ *Developer's Guide*	Developers	Provides a quick-start tutorial and programming information relevant to the MQ implementation of JMS.
MQ *Administrator's Guide*	Administrators, also recommended for developers	Provides background and information needed to perform administration tasks using MQ administration tools.

Online Help

MQ 3.0.1 includes command line utilities for performing MQ message service administration tasks. To access the online help for these utilities, see "Common Command Line Options" on page 87.

MQ 3.0.1 also includes a graphical user interface (GUI) administration tool, the Administration Console (`imqadmin`). Context sensitive online help is included in the Administration Console.

JavaDoc

JMS and MQ API documentation in JavaDoc format, is provided at the following location:

`IMQ_HOME/javadoc/index.html`
(`/usr/share/javadoc/imq/index.html` on Solaris)

This documentation can be viewed in any HTML browser such as Netscape or Internet Explorer. It includes standard JMS API documentation as well as MQ-specific API's for MQ administered objects (see Chapter 3 of the MQ *Developer's Guide*), which are of value to developers of messaging applications.

Example Client Applications

A number of example applications that provide sample client application code are included in the following location:

`IMQ_HOME/demo` (`/usr/demo/imq` on Solaris)

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

The Java Message Service (JMS) Specification

The JMS specification can be found at the following location:

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

The specification includes sample client code.

Chapter 1

Overview

This chapter provides an introduction to Sun™ ONE Message Queue (MQ) and is of interest to both administrators and programmers.

What Is Sun ONE Message Queue?

The MQ product is a standards-based solution to the problem of inter-application communication and reliable message delivery. MQ is an enterprise messaging system that implements the Java Message Service (JMS) open standard: it is a JMS provider.

The JMS specification describes a set of programming interfaces (see "JMS Programming Model" on page 32)—which provide a common way for Java applications to create, send, receive, and read messages in a distributed environment.

With Sun ONE Message Queue software, processes running on different platforms and operating systems can connect to a common MQ message service (see "Message Service Architecture" on page 30) to send and receive information. Application developers are free to focus on the business logic of their applications, rather than on the low-level details of how their applications communicate across a network.

MQ has features which exceed the minimum requirements of the JMS specification. Among these features are the following:

Centralized administration Provides both command-line and GUI tools for administering an MQ message service and managing application-specific aspects of messaging, such as destinations and security.

Scalable message service Allows you to service increasing numbers of JMS clients (components or applications) by balancing the load among a number of MQ message service components (*brokers*) working in tandem (multi-broker cluster).

Tunable performance Lets you increase performance of the MQ message service when less reliability of delivery is acceptable.

Multiple transports Supports the ability of JMS clients to communicate with each other over a number of different transports, including TCP and HTTP, and using secure (SSL) connections.

JNDI support Supports both file-based and LDAP implementations of the Java Naming and Directory Interface (JNDI) as object stores and user repositories.

SOAP messaging support Supports creation and delivery of SOAP messages—messages that conform to the Simple Object Access Protocol (SOAP) specification— *via* JMS messaging. SOAP allows for the exchange of structured XML data between peers in a distributed environment. See the MQ *Developer's Guide* for more information.

See the MQ 3.0.1 *Release Notes* for documentation of JMS compliance-related issues.

Product Editions

The Sun ONE Message Queue product is available in two editions: Platform and Enterprise—each corresponding to a different licensed capacity, as described below. (To upgrade MQ from one edition to another, see the instructions in the MQ *Installation Guide*.)

Platform Edition

This edition can be downloaded free from the Sun website and is also bundled with the latest Sun ONE Application Server platform. The Platform Edition places no limit on the number of JMS client connections supported by each MQ message service. It comes with two licenses, as described below:

- **a basic license** This license provides basic JMS support (it's a full JMS provider), but does not include such enterprise features as load balancing (multi-broker message service), HTTP/HTTPS connections, secure connection services, scalable connection capability, and multiple queue delivery policies. The license has an unlimited duration, and can therefore be used in less demanding production environments.

- **a 90-day trial enterprise license** This license includes all enterprise features (such as support for multi-broker message services, HTTP/HTTPS connections, secure connection services, scalable connection capability, and multiple queue delivery policies) not included in the basic license. However, the license has a limited 90-day duration enforced by the software, making it suitable for evaluating the enterprise features available in the Enterprise Edition of the product (see "Enterprise Edition" on page 27).

NOTE The 90-day trial license can be enabled by starting the MQ message service—an MQ broker instance—with the `-license` command line option (see Table 5-2 on page 121) and passing "`try`" as the license to use:

```
imqbrokerd -license try
```

You must use this option each time you start the broker instance, otherwise it defaults back to the basic Platform Edition license.

Enterprise Edition

This edition is for deploying and running messaging applications in a production environment. It includes support for multi-broker message services, HTTP/HTTPS connections, secure connection services, scalable connection capability, and multiple queue delivery policies. You can also use the Enterprise Edition for developing, debugging, and load testing messaging applications and components. The Enterprise Edition has an unlimited duration license that places no limit on the number of brokers in a multi-broker message service, but specifies the number of CPU's that are supported.

Enterprise Messaging Systems

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

Requirements of Enterprise Messaging Systems

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

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

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

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

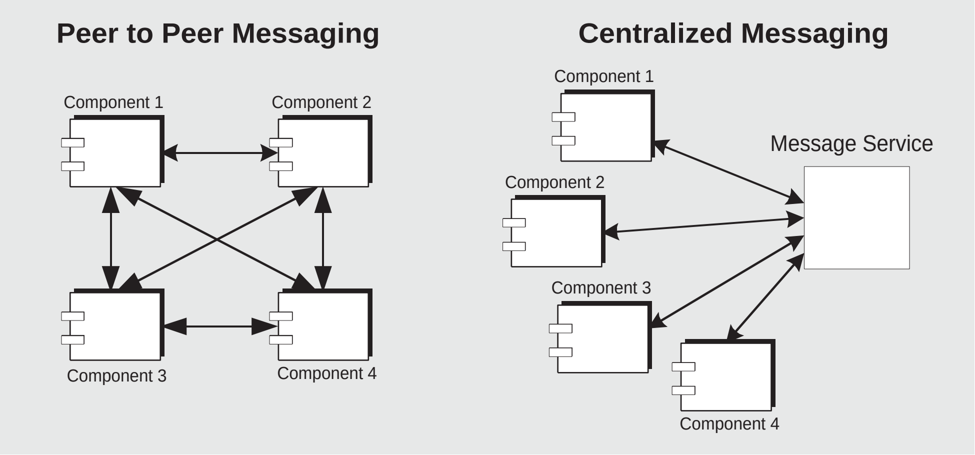
Scalability The messaging system must be able to accommodate increasing loads—increasing numbers of users and increasing numbers of messages—without a substantial loss of performance or message throughput. As businesses and applications expand, this becomes a very important requirement.

Manageability The messaging system must provide tools for monitoring and managing the delivery of messages and for optimizing system resources. These tools help measure and maintain reliability, security, and performance.

Centralized vs. Peer to Peer Messaging

The requirements of an enterprise messaging system are difficult to meet with a traditional peer to peer messaging system, illustrated in Figure 1-1.

Figure 1-1 Centralized vs. Peer to Peer Messaging



In such a system every messaging component maintains a connection to every other component. These connections can allow for fast, secure, and reliable delivery, however the code for supporting reliability and security must reside in each component. As components are added to the system, the number of connections rises exponentially. This makes asynchronous message delivery and scalability difficult to achieve. Centralized management is also problematic.

The preferred approach for enterprise messaging is a centralized messaging system, also illustrated in Figure 1-1. In this approach each messaging component maintains a connection to one central message service. The message service provides for routing and delivery of messages between components, and is responsible for reliable delivery and security. Components interact with the message service through a well-defined programming interface. As components are added to the system, the number of connections rises only linearly, making it easier to scale the system by scaling the message service. In addition, the central message service provides for centralized management of the system.

Messaging System Concepts

A few basic concepts underlie enterprise messaging systems. These include the following: message, message service architecture, and message delivery models.

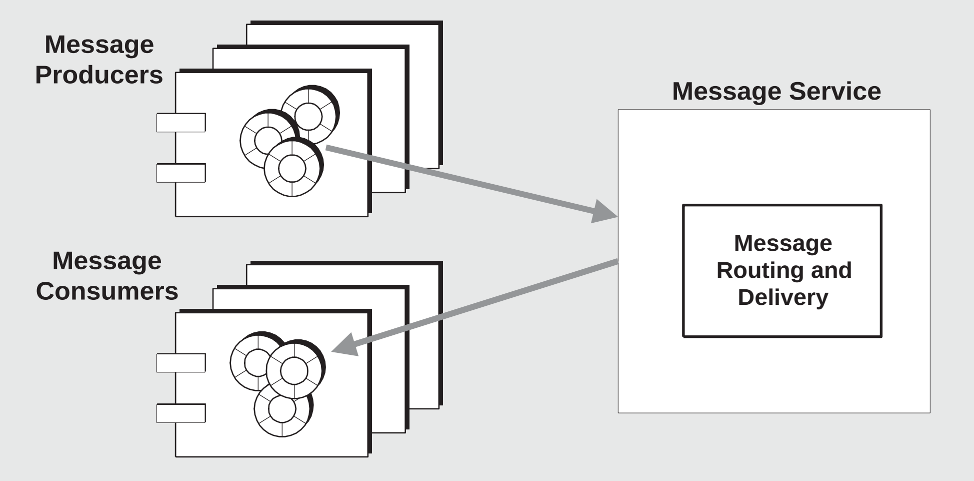
Message

A message consists of data in some format (message body) and meta-data that describes the characteristics or properties of the message (message header), such as its destination, lifetime, or other characteristics determined by the messaging system.

Message Service Architecture

The basic architecture of a messaging system is illustrated in Figure 1-2 on page 30. It consists of message producers and message consumers that exchange messages by way of a common message service. Any number of message producers and consumers can reside in the same messaging component (or application). A message producer sends a message to a message service. The message service, in turn, using message routing and delivery components, delivers the message to one or more message consumers that have registered an interest in the message. The message routing and delivery components are responsible for guaranteeing delivery of the message to all appropriate consumers.

Figure 1-2 Message Service Architecture



Message Delivery Models

There are many relationships between producers and consumers: one to one, one to many, and many to many relationships. For example, you might have messages delivered from:

- one producer to one consumer
- one producer to many consumers
- many producers to one consumer
- many producers to many consumers.

These relationships are often reduced to two message delivery models: *point-to-point* and *publish/subscribe* messaging. The focus of the point-to-point delivery model is on messages that originate from a specific producer and are received by a specific consumer. The focus of publish/subscribe delivery model is on messages that originate from any of a number of producers and are received by any number of consumers. These message delivery models can overlap.

Historically, messaging systems supported various combinations of these two message delivery models. The Java Message Service (JMS) API was intended to create a common programming approach for Java messaging. It supports both the point-to-point and publish/subscribe message delivery models (see "Programming Domains" on page 37).

The JMS Specification

JMS specifies a message structure, a programming model, and a set of rules and semantics that govern messaging operations. Because MQ provides an implementation of JMS, JMS concepts are fundamental to understanding how an MQ messaging system works. This introduction explains concepts and terminology needed to understand the remaining chapters of this book.

JMS Message Structure

According to the JMS specification, a message is composed of three parts: a header, properties, and a body.

Header The header specifies the JMS characteristics of the message: its destination, whether it is persistent or not, its time to live, and its priority. These characteristics govern how the messaging system delivers the message.

Properties Properties (which can be thought of as an extension of the header) are optional—they provide values that applications can use to filter messages according to various selection criteria. Properties are optional.

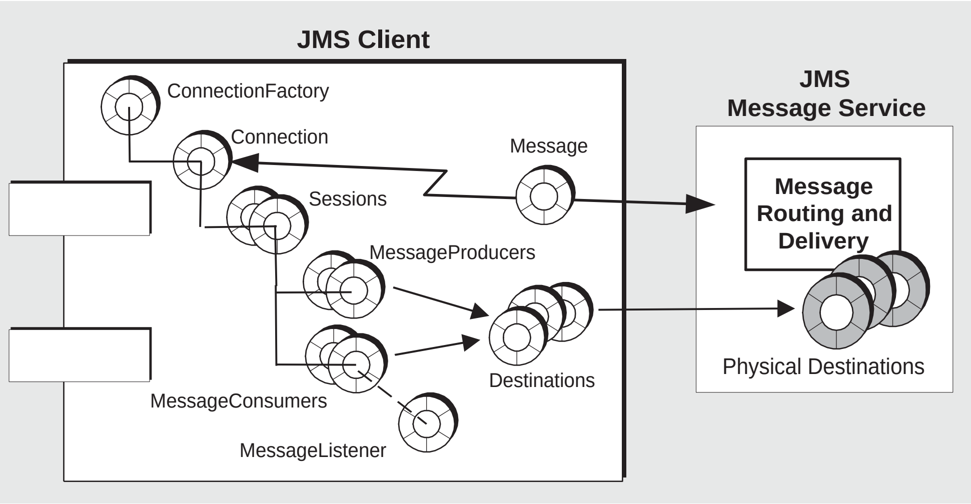
Message body. The message body contains the actual data to be exchanged. JMS supports six body types.

JMS Programming Model

In the JMS programming model, JMS clients (components or applications) exchange messages by way of a JMS message service. Message producers send messages to the message service, from which message consumers receive them. These messaging operations are performed using a set of objects (furnished by a JMS provider) that implement the JMS application programming interface (API).

This section introduces the objects that implement the JMS API and that are used to set up a JMS client for delivery of messages (for more information, see the MQ *Developer's Guide*). Figure 1-3 on page 32 shows the JMS objects used to program the delivery of messages.

Figure 1-3 JMS Programming Objects



In the JMS programming model, a JMS client uses a `ConnectionFactory` object to create a connection over which messages are sent to and received from the JMS message service. A `Connection` is a JMS client's active connection to the message service. Both allocation of communication resources and authentication of the client take place when a connection is created. It is a relatively heavy-weight object, and most clients do all their messaging with a single connection.

The connection is used to create sessions. A `Session` is a single-threaded context for producing and consuming messages. It is used to create the message producers and consumers that send and receive messages, and it defines a serial order for the messages it delivers. A session supports reliable delivery through a number of acknowledgement options or through transactions (which can be managed by a distributed transaction manager).

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

Similarly, a JMS client uses a `MessageConsumer` to receive messages from a specified physical destination, represented in the API as a destination object. A message consumer can use a message selector that allows the message service to deliver only those messages to the message consumer that match the selection criteria.

A message consumer can support either synchronous or asynchronous consumption of messages (see the MQ *Developer's Guide*). Asynchronous consumption is achieved by registering a `MessageListener` with the consumer. The client consumes a message when a session thread invokes the `onMessage()` method of the `MessageListener` object.

Administered Objects

Two of the objects described in the "JMS Programming Model" on page 32 depend on how a JMS provider implements a JMS message service. The connection factory object depends on the underlying protocols and mechanisms used by the provider to deliver messages, and the destination object depends on the specific naming conventions and capabilities of the physical destinations used by the provider.

Normally these provider-specific characteristics would make JMS client code dependent on a specific JMS implementation. To make JMS client code provider-independent, however, the JMS specification requires that provider-specific implementation and configuration information be encapsulated in what are called *administered objects.* These objects can then be accessed in a standardized, non-provider-specific way.

Administered objects are created and configured by an administrator, stored in a name service, and accessed by JMS clients through standard Java Naming and Directory Service (JNDI) lookup code. Using administered objects in this way makes JMS client code provider-independent.

JMS provides for two general types of administered objects: connection factories and destinations. Both encapsulate provider-specific information, but they have very different uses within a JMS client. A connection factory is used to create connections to a message server, while destination objects are used to identify physical destinations used by the JMS message service.

For more information on administered objects, see "MQ Administered Objects" on page 75.

JMS/J2EE Programming: Message-driven Beans

In addition to the general JMS client programming model introduced in "JMS Programming Model" on page 32, there is a more specialized adaptation of JMS used in the context of Java 2 Enterprise Edition (J2EE) applications. This specialized JMS client is called a *message-driven bean* and is one of a family of Enterprise JavaBeans (EJB) components specified in the EJB 2.0 Specification (`http://java.sun.com/products/ejb/docs.html`).

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

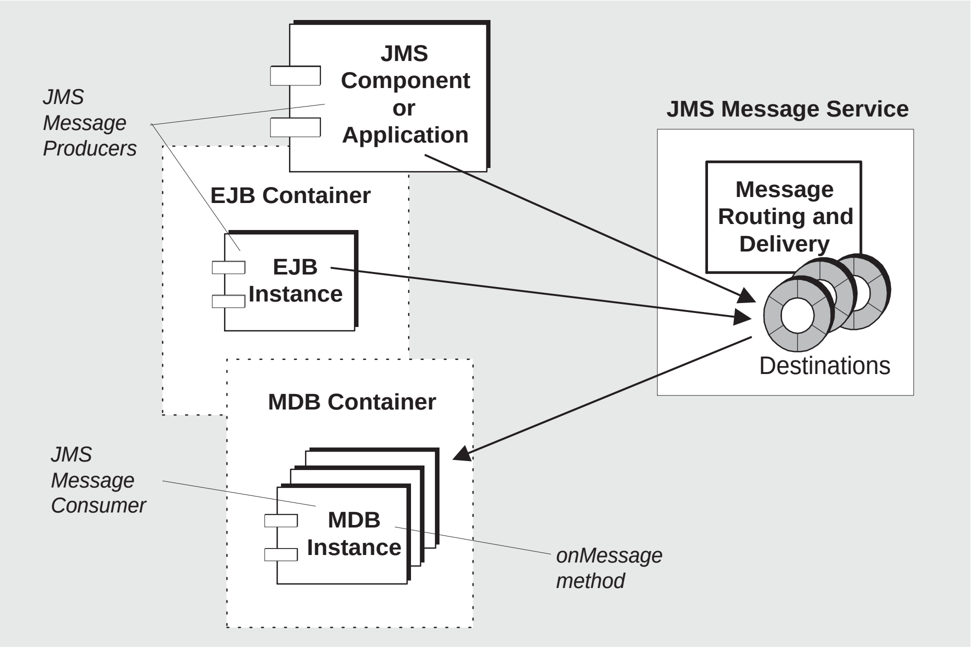
However, asynchronous messaging is a requirement of many enterprise applications. Most such applications require that server-side components be able to communicate and respond to each other without tying up server resources. Hence, the need for an EJB component that can receive messages and consume them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

Message-driven Beans

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

Message-driven Bean The MDB is a JMS message consumer that implements the JMS `MessageListener` interface. The `onMessage` method (written by the MDB developer) is invoked when a message is received by the MDB container. The `onMessage()` method consumes the message, just as the `onMessage()` method of a standard `MessageListener` object would. You do not remotely invoke methods on MDB's—like 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 1-4 on page 35.

Figure 1-4 Messaging with MDBs



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

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

Application Server Support

In J2EE architecture (see the J2EE Platform Specification located at `http://java.sun.com/j2ee/download.html#platformspec`), EJB containers are hosted by 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 MDB's, a JMS provider.

In the Sun ONE Application Server, messaging resources are provided by Sun ONE Message Queue. This means that an MQ messaging system (see Chapter 2, "The MQ Messaging System") is integrated into the Sun ONE Application Server, providing the support needed to send JMS messages to MDB's and other JMS messaging components that run in the application server environment.

JMS Messaging Issues

This section describes a number of JMS programming issues that impact the administration of an MQ message service. The discussion focuses on concepts and terminology that are needed by an MQ administrator.

JMS Provider Independence

JMS specifies the use of administered objects (see "Administered Objects" on page 33) to support the development of client applications that are portable to other JMS providers. Administered objects allow JMS clients to use logical names to look up and reference provider-specific objects. In this way client code does not need to know specific naming or addressing syntax or configurable properties used by a provider. This makes the code provider-independent.

Administered objects are MQ system objects created and configured by an MQ administrator. These objects are placed in a JNDI directory service, and a JMS client accesses them using a JNDI lookup.

MQ administered objects can also be instantiated by the client, rather than looked up in a JNDI directory service. This has the drawback of requiring the application developer to use provider-specific API's. It also undermines the ability of an MQ administrator to successfully control and manage an MQ message server.

For more information on administered objects, see "MQ Administered Objects" on page 75.

Programming Domains

JMS supports two distinct message delivery models: point-to-point and publish/subscribe.

point-to-point (Queue Destinations) A message is delivered from a producer to one consumer. In this delivery model, the destination is a *queue*. Messages are first delivered to the queue destination, then delivered from the queue, one at a time, depending on the queue's delivery policy (see "Queue Destinations" on page 65), to one of the consumers registered for the queue. Any number of producers can send messages to a queue destination, but each message is guaranteed to be delivered to—and successfully consumed by—only *one* consumer. If there are no consumers registered for a queue destination, the queue holds messages it receives, and delivers them when a consumer registers for the queue.

Publish/Subscribe (Topic destinations) A message is delivered from a producer to any number of consumers. In this delivery model, the destination is a *topic*. Messages are first delivered to the topic destination, then delivered to *all* active consumers that have *subscribed* to the topic. Any number of producers can send messages to a topic destination, and each message can be delivered to any number of subscribed consumers. Topic destinations also support the notion of *durable subscriptions.* A durable subscription represents a consumer that is registered with the topic destination but can be inactive at the time that messages are delivered. When the consumer subsequently becomes active, it receives the messages. If there are no consumers registered for a topic destination, the topic does not hold messages it receives, unless it has durable subscriptions for inactive consumers.

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

Table 1-1 JMS Programming Objects

Base Type (Unified Domain)	Point-to-Point Domain	Publish/Subscribe Domain
Destination (Queue or Topic)[1]	Queue	Topic
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

[1] Depending on programming approach, you might specify a particular destination type.

You can program both point-to-point and publish/subscribe messaging using the unified domain objects shown in the first column of Table 1-1. This is the preferred approach. However, to conform to the earlier JMS 1.02b specification, you can use the point-to-point domain objects to program point-to-point messaging, and the publish/subscribe domain objects to program publish/subscribe messaging.

Client Identifiers

JMS providers must support the notion of a *client identifier*, which associates a JMS client's connection to a message service with state information maintained by the message service on behalf of the client. By definition, a client identifier is unique, and applies to only one user at a time. Client identifiers are used in combination with a durable subscription name (see "Publish/Subscribe (Topic destinations)" on page 37) to make sure that each durable subscription corresponds to only one user.

The JMS specification allows client identifiers to be set by the client through an API method call, but recommends setting it administratively using a connection factory administered object (see "Administered Objects" on page 33). If hard wired into a connection factory, however, each user would then need an individual connection factory to have a unique identity.

MQ provides a way for the client identifier to be both ConnectionFactory and user specific using a special variable substitution syntax that you can configure in a `ConnectionFactory` object. When used this way, a single `ConnectionFactory` object can be used by multiple users who create durable subscriptions, without fear of naming conflicts or lack of security. A user's durable subscriptions are therefore protected from accidental erasure or unavailability due to another user having set the wrong client identifier.

For details on how to use this MQ feature, see the discussion of connection factory attributes in the MQ *Developer's Guide*.

In any case, in order to create a durable subscription, a client identifier must be either programmatically set by the client, using the JMS API, or administratively configured in the `ConnectionFactory` objects used by the client.

Reliable Messaging

JMS defines two *delivery modes*:

Persistent messages These messages are guaranteed to be delivered and successfully consumed once and only once. Reliability is at a premium for such messages.

Non-persistent messages These messages are guaranteed to be delivered at most once. Reliability is not a major concern for such messages.

There are two aspects of assuring reliability in the case of *persistent* messages. One is to assure that their delivery to and from a message service is successful. The other is to assure that the message service does not lose persistent messages before delivering them to consumers.

Acknowledgements/Transactions

Reliable messaging depends on guaranteeing the successful delivery of persistent messages to and from a destination. This can be achieved using either of two general mechanisms supported by an MQ session: acknowledgements or transactions. In the case of transactions, these can either be local or distributed, under the control of a distributed transaction manager.

Acknowledgements

A session can be configured to use acknowledgements to assure reliable delivery.

In the case of a producer, this means that the message service acknowledges delivery of a persistent message to its destination before the producer's `send()` method returns. In the case of a consumer, this means that the client acknowledges delivery and consumption of a persistent message from a destination before the message service deletes the message from that destination.

Local Transactions

A session can also be configured as *transacted*, in which case the production and/or consumption of one or more messages can be grouped into an atomic unit—a *transaction*. The JMS API provides methods for initiating, committing, or rolling back a transaction.

As messages are produced or consumed within a transaction, the broker tracks the various sends and receives, completing these operations only when the 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 the successful operations are completed. When a transaction is rolled back, all successful operations are cancelled.

The scope of a local transaction is always a single session. That is, one or more producer or consumer operations performed in the context of a single session can be grouped into a single local transaction.

Since transactions span only a single session, you cannot have an end-to-end transaction encompassing both the production and consumption of a message. (In other words, the delivery of a message to a destination and the subsequent delivery of the message to a client cannot be placed in a single transaction.)

Distributed Transactions

MQ also supports *distributed* transactions. That is, the production and consumption of messages can be part of a larger, distributed transaction that includes operations involving other resource managers, such as database systems. In distributed transactions, a distributed transaction manager tracks and manages operations performed by multiple resource managers (such as a message service and a database manager) using a two-phase commit protocol defined in the Java Transaction API (JTA), *XA Resource* API specification. In the Java world, interaction between resource managers and a distributed transaction manager are described in the JTA specification.

Support for distributed transactions means that messaging clients can participate in distributed transactions through the XAResource interface defined by JTA. This interface defines a number of methods for implementing two-phase commit. While the API calls are made on the client side, the MQ broker 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.

MQ implements support for distributed transactions through an XA connection factory, which lets you create XA connections, which in turn lets you create XA sessions (see "JMS Programming Model" on page 32). In addition, support for distributed transactions requires either a third party JTS or a J2EE-compliant Application Server (that provides JTS).

Persistent Storage

The other important aspect of reliability is assuring that once persistent messages are delivered to their destinations, a message service does not lose them before they are delivered to consumers. This means that upon delivery of a persistent message to its destination, the message service must place it in a persistent data store (see "Persistence Manager" on page 54). If the message service goes down for any reason, it can recover the message and deliver it to the appropriate consumers. While this adds overhead to message delivery, it also adds reliability.

A message service must also store durable subscriptions. This is because to guarantee delivery in the case of topic destinations, it is not sufficient to recover only persistent messages. The message service must also recover information about durable subscriptions for a topic, otherwise it would not be able to deliver messages to subscribers who are inactive when a message arrives, and subsequently become active.

Messaging applications that are concerned about guaranteed message delivery must specify messages as persistent and use either queue destinations or durable subscriptions to topic destinations.

Performance Trade-offs

The more reliable the delivery of messages, the more overhead and bandwidth are required to achieve it. The trade-off between reliability and performance is a significant design consideration. You can maximize *performance* by choosing to produce and consume non-persistent messages. On the other hand, you can maximize *reliability* by producing and consuming persistent messages and using transacted sessions. Between these extremes are a number of options, depending on the needs of an application, including the use of MQ-specific connection and acknowledgement properties (see the MQ *Developer's Guide*).

Message Selection

JMS provides a mechanism by which a message service can perform message filtering and routing based on criteria placed in message selectors. A producing client can place application-specific properties in the message, and a consuming client can indicate its interest in messages using selection criteria based on such properties. This simplifies the work of the client and eliminates the overhead of delivering messages to clients that don't need them. However, it adds some additional overhead to the message service processing the selection criteria. Message selector syntax and semantics are outlined in the JMS specification.

Message Order and Priority

In general, all messages sent to a destination by a single session are guaranteed to be delivered to a consumer in the order they were sent. However, if they are assigned different priorities, a messaging system will attempt to deliver higher priority messages first.

Beyond this, the ordering of messages consumed by a client application can have only a rough relationship to the order in which they were produced. This is because the delivery of messages to destinations and the delivery from those destinations can depend on a number of issues that affect timing, such as the order in which the messages are sent, the sessions (connections) from which they are sent, whether the messages are persistent, the lifetime of the messages, the priority of the messages, the message delivery policy of queue destinations (see "Queue Destinations" on page 65), and message service availability.

In the case of an MQ message server using multiple interconnected brokers (see "Multi-Broker Clusters (Enterprise Edition)" on page 68) the ordering of messages consumed by a client is further complicated by the fact that the order of delivery from destinations on different brokers is indeterminate. Hence, a message delivered by one broker might precede a message delivered by another broker even though the latter might have received the message first.

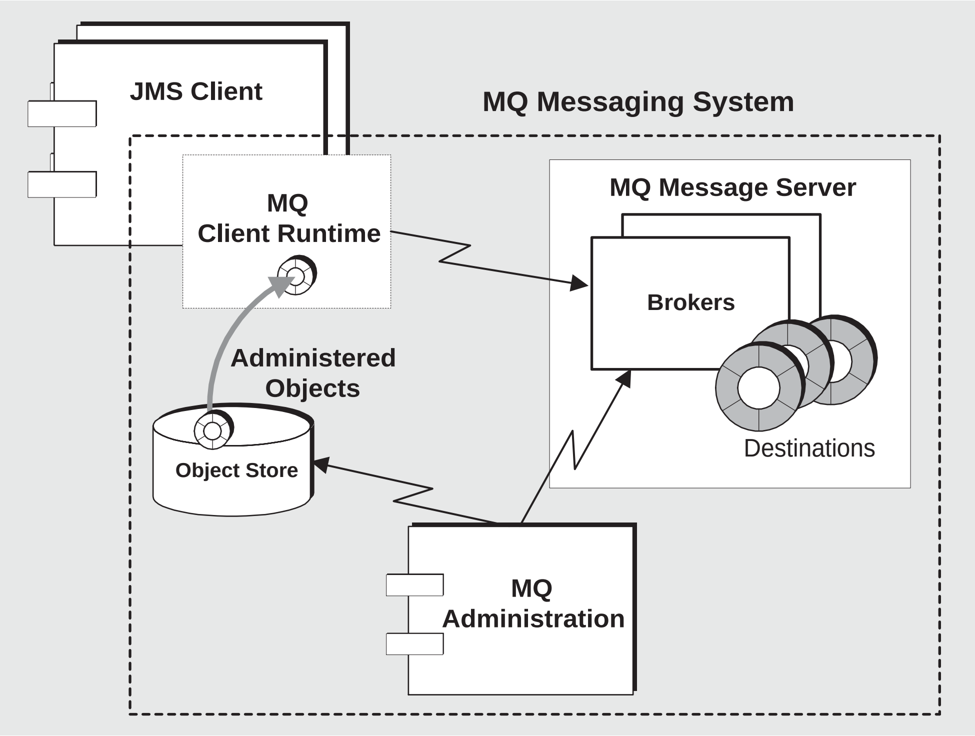
In any case, for a given consumer, precedence is given for higher priority messages over lower priority messages.

Chapter 2

The MQ Messaging System

This chapter describes the Sun™ ONE Message Queue (MQ) messaging system, with specific attention to the main parts of the system, as illustrated in Figure 2-1, and explains how they work together to provide for reliable message delivery.

Figure 2-1 MQ System Architecture



The main parts of an MQ messaging system, shown in Figure 2-1, are the following:

- MQ Message Server
- MQ Client Runtime
- MQ Administered Objects
- MQ Administration

The first three of these are examined in the following sections. The last is introduced in Chapter 3, "MQ Administration."

MQ Message Server

This section describes the different parts of the MQ message server shown in Figure 2-1 on page 43. These include the following:

Broker An MQ broker provides delivery services for an MQ messaging system. Message delivery relies upon a number of supporting components that handle connection services, message routing and delivery, persistence, security, and logging (see "Broker" on page 44 for more information). A message server can employ one or more broker instances (see "Multi-Broker Clusters (Enterprise Edition)" on page 68).

Physical Destination Delivery of a message is a two-phase process—delivery from a producing client to a physical destination maintained by a broker, followed by delivery from the destination to one or more consuming clients. Physical destinations represent locations in a broker's physical memory and/or persistent storage (see "Physical Destinations" on page 65 for more information).

Broker

Message delivery in an MQ messaging system—from producing clients to destinations, and then from destinations to one or more consuming clients—is performed by a broker (or a cluster of broker instances working in tandem). To perform message delivery, a broker must set up communication channels with clients, perform authentication and authorization, route messages appropriately, guarantee reliable delivery, and provide data for monitoring system performance.

To perform this complex set of functions, a broker uses a number of different components, each with a specific role in the delivery process. You can configure these internal components to optimize the performance of the broker, depending on load conditions, application complexity, and so on. The main broker components are illustrated in Figure 2-2 and described briefly in Table 2-1.

Figure 2-2 Broker Components

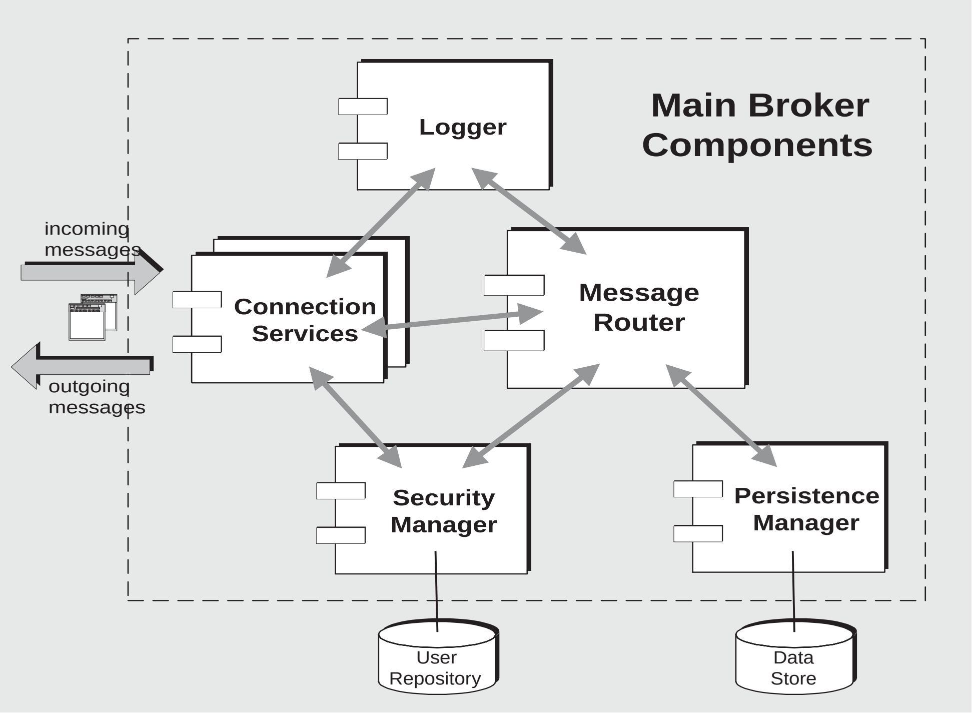


Table 2-1 Main Broker Components and Functions

Component	Description/Function
Connection Services	Manages the physical connections between a broker and clients, providing transport for incoming and outgoing messages.
Message Router	Manages the routing and delivery of messages: These include JMS messages as well as control messages used by the MQ messaging system to support JMS message delivery.

Table 2-1 Main Broker Components and Functions *(Continued)*

Component	Description/Function
Persistence Manager	Manages the writing of data to persistent storage so that system failure does not result in failure to deliver JMS messages.
Security Manager	Provides authentication services for users requesting connections to a broker and authorization services (access control) for authenticated users.
Logger	Writes monitoring and diagnostic information to log files or the console so that an administrator can monitor and manage a broker.

The following sections explore more fully the functions performed by the different broker components and the properties that can be configured to affect their behavior.

Connection Services

An MQ broker supports communication with both JMS clients and MQ administration clients (see "MQ Administration Tools" on page 84). Each service is specified by its service type and protocol type.

service type specifies whether the service provides JMS message delivery (`NORMAL`) or MQ administration (`ADMIN`) services

protocol type specifies the underlying transport protocol layer that supports the service.

The connection services currently available from an MQ broker are shown in Table 2-2:

Table 2-2 Connection Services Supported by a Broker

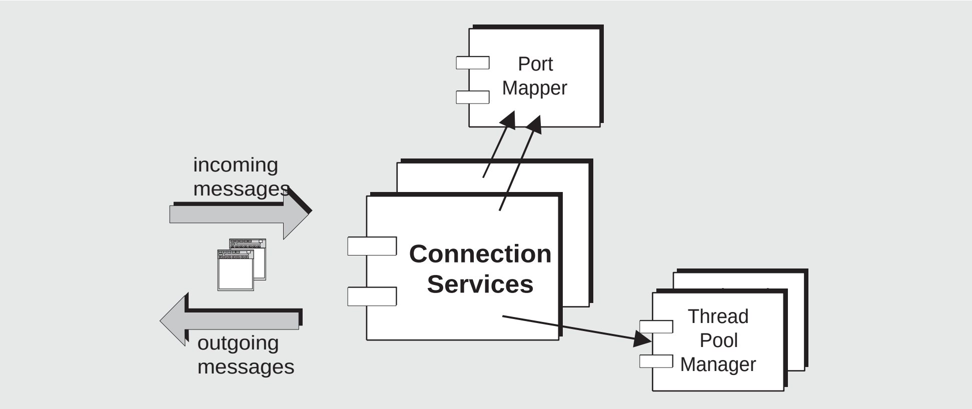
Service Name	Service Type	Protocol Type
jms	NORMAL (JMS message delivery)	tcp
ssljms (Enterprise Edition)	NORMAL (JMS message delivery)	tls (SSL-based security)

Table 2-2 Connection Services Supported by a Broker *(Continued)*

Service Name	Service Type	Protocol Type
httpjms (Enterprise Edition)	NORMAL (JMS message delivery)	http
httpsjms (Enterprise Edition)	NORMAL (JMS message delivery)	https (SSL-based security)
admin	ADMIN	tcp
ssladmin (Enterprise Edition)	ADMIN	tls (SSL-based security)

You can configure a broker to run any or all of these connection services. Each service has a Thread Pool Manager and registers itself with a common Port Mapper service, as shown in Figure 2-3.

Figure 2-3 Connection Services Support



Each connection service is available at a particular port, specified by the broker's host name and a port number. The port can be statically or dynamically allocated. MQ provides a *Port Mapper* that maps dynamically allocated ports to the different connection services. The Port Mapper itself resides at a standard port number, 7676. When a client sets up a connection with the broker, it first contacts the Port Mapper requesting the port number of the connection service it desires.

You can also assign a *static* port number for the jms, ssljms, admin and ssladmin connection services when configuring these connection services, but this is not recommended. The httpjms and httpsjms services are configured using properties described in Table B-1 on page 217 and Table B-3 on page 226, respectively, in Appendix B, "HTTP/HTTPS Support (Enterprise Edition)."

Each connection service is multi-threaded, supporting multiple connections. The threads needed for these connections are maintained in a thread pool managed by a *Thread Pool Manager* component. You can configure the Thread Pool Manager to set a minimum number and maximum number of threads maintained in the thread pool. As threads are needed by connections, they are added to the thread pool. When the minimum number is exceeded, the system will shut down threads as they become free until the minimum number threshold is reached, thereby saving on memory resources. You want this number to be large enough so that new threads do not have to be continually created. Under heavy connection loads, the number of threads might increase until the thread pool's maximum number is reached, after which connections have to wait until a thread becomes available.

The threads in a thread pool can either be dedicated to a single connection (*dedicated* model) or assigned to multiple connections, as needed (*shared* model).

Dedicated model In the dedicated model, each connection to the broker requires two threads: one dedicated to handling incoming messages and one to handling outgoing messages. This limits the number of connections to half the maximum number of threads in the thread pool, however it provides for high performance.

Shared model (Enterprise Edition) In the shared thread model, connections are assigned to a thread only when sending or receiving messages. This model, in which connections share a thread, increases the number of connections that a connection service (and therefore, a broker) can support, however there is some performance overhead involved. The Thread Pool Manager uses a set of distributor threads that monitor connection activity and assign connections to threads as needed. You can improve performance by limiting the number of connections monitored by each such distributor thread.

Each connection service supports specific authentication and authorization (access control) features (see "Security Manager" on page 57).

The configurable properties related to connection services are shown in Table 2-3. (For instructions on configuring these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 2-3 Connection Service Properties

Property Name	Description
`imq.service.activelist`	List of connection services, by name, separated by commas, to be made active at broker startup. Supported services are: `jms`, `ssljms`, `httpjms`, `httpsjms`, `admin`, `ssladmin`. Default: `jms`, `admin`
`imq.`*service_name*`.min_threads`	Specifies the number of threads, which once reached, are maintained in the thread pool for use by the named connection service. Default: Depends on connection service (see Table 5-1 on page 116).
`imq.`*service_name*`.max_threads`	Specifies the number of threads beyond which no new threads are added to the thread pool for use by the named connection service. The number must be greater than zero and greater in value than the value of `min_threads`. Default: Depends on connection service (see Table 5-1 on page 116).
`imq.`*service_name*`.threadpool_model`	Specifies whether threads are dedicated to connections (`dedicated`) or shared by connections as needed (`shared`) for the named connection service. Shared model (threadpool management) increases the number of connections supported by a broker, but is implemented only for the `jms` and `admin` connection services. Default: Depends on connection service (see Table 5-1 on page 116).
`imq.shared.connectionMonitor_limit`	For shared threadpool model only, specifies the maximum number of connections that can be monitored by a distributor thread. (The system allocates enough distributor threads to monitor all connections.) The smaller this value, the faster the system can assign active connections to threads. A value of `0` means no limit. Default: Depends on operating system (see Table 5-1 on page 116).
`imq.portmapper.port`	The broker's primary port—the port at which the Port Mapper resides. If you are running more than one broker instance on a host, each must be assigned a unique Port Mapper port. Default: `7676`

Table 2-3 Connection Service Properties *(Continued)*

Property Name	Description
`imq.`*service_name*`.` *protocol_type*[1]`.port`	For jms, ssljms, admin, and ssladmin services only, specifies the port number for the named connection service. Default: `0` (port is dynamically allocated by the Port Mapper) To configure the httpjms and httpsjms connection services, see Appendix B, "HTTP/HTTPS Support (Enterprise Edition)."
`imq.`*service_name*`.` *protocol_type*[1]`.hostname`	For jms, ssljms, admin, and ssladmin services only, specifies the host (hostname or IP address) to which the named connection service binds if there is more than one host available (for example, if there is more than one network interface card in a computer). Default: `null` (any host)

[1] protocol_type is specified in Table 2-2.

Message Router

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

Basic Delivery Mechanisms

Broadly speaking, the messages handled by a broker fall into two categories: the JMS messages sent by producer clients, destined for consumer clients—payload messages, and a number of control messages that are sent to and from clients in order to support the delivery of the JMS messages.

If the incoming message is a JMS message, the broker routes it to consumer clients, based on the type of its destination (queue or topic):

- If the destination is a topic, the JMS message is immediately routed to all active subscribers to the topic. In the case of inactive durable subscribers, the Message Router holds the message until the subscriber becomes active, and then delivers the message to that subscriber.
- If the destination is a queue, the JMS message is placed in the corresponding queue, and delivered to the appropriate consumer when the message reaches the front of the queue. The order in which messages reach the front of the queue depends on the order of their arrival and on their priority.

Once the Message Router has delivered a message to all its intended consumers it clears the message from memory, and if the message is persistent (see "Reliable Messaging" on page 39), removes it from the broker's persistent data store.

Reliable Delivery: Acknowledgements, and Transactions

The delivery mechanism just described becomes more complicated when adding requirements for *reliable* delivery (see "Reliable Messaging" on page 39). There are two aspects involved in reliable delivery: assuring that delivery of messages to and from a broker is successful, and assuring that the broker does not lose messages or delivery information before messages are actually delivered.

To ensure that messages are successfully delivered to and from a broker, MQ uses a number of control messages called acknowledgements.

For example, when a producer sends a JMS message (a payload message as opposed to a control message) to a destination, the broker sends back a control message—a broker acknowledgement—that it received the JMS message. (In practice, MQ only does this if the producer specifies the JMS message as persistent.) The producing client uses the broker acknowledgement to guarantee delivery to the destination (see "Message Production" on page 74).

Similarly, when a broker delivers a JMS message to a consumer, the consuming client sends back an acknowledgement that it has received and processed the message. A client specifies how automatically or how frequently to send these acknowledgments when creating session objects, but the principle is that the Message Router will not delete a JMS message from memory if it has not received an acknowledgement from each message consumer to which it has delivered the message—for example, from each of the multiple subscribers to a topic.

In the case of durable subscribers to a topic, the Message Router retains each JMS message in that destination, delivering it as each durable subscriber becomes an active consumer. The Message Router records client acknowledgements as they are received, and deletes the JMS message only after all the acknowledgements have been received (unless the JMS message expires before then).

Furthermore, the Message Router confirms receipt of the client acknowledgement by sending a broker acknowledgement back to the client. The consuming client uses the broker acknowledgement to make sure that the broker will not deliver a JMS message more than once (see "Message Consumption" on page 74). This could happen if, for some reason, the broker fails to receive the client acknowledgement).

If the broker does not receive a client acknowledgement and re-delivers a JMS message a second time, the message is marked with a Redeliver flag. The broker generally re-delivers a JMS message if a client connection closes before the broker receives a client acknowledgement, and a new connection is subsequently opened. For example, if a message consumer of a queue goes off line before acknowledging a message, and another consumer subsequently registers with the queue, the broker will re-deliver the unacknowledged message to the new consumer.

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

The broker tracks transactions, allowing them to be committed or rolled back should they fail. This transaction management also supports local transactions that are part of larger, distributed transactions (see "Distributed Transactions" on page 40). The broker tracks the state of these transactions until they are committed. When a broker starts up it inspects all uncommitted transactions and, by default, rolls back all transactions except those in a `PREPARED` state.

Reliable Delivery: Persistence

The other aspect of reliable delivery is assuring that the broker does not lose messages or delivery information before messages are actually delivered. In general, messages remain in memory until they have been delivered or they expire. However, if the broker should fail, these messages would be lost.

A producer client can specify that a message be persistent, and in this case, the Message Router will pass the message to a *Persistence Manager* that stores the message in a database or file system (see "Persistence Manager" on page 54) so that the message can be recovered if the broker fails.

Managing System Resources

The performance of a broker depends on the system resources available and how efficiently resources such as memory are utilized. For example, the Message Router has a memory management scheme that watches memory on the system. When memory resources become scarce, mechanisms for reclaiming memory and for slowing the flow of incoming messages are activated.

The memory management mechanism 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 state of memory resources progresses from `green` through `yellow` and `orange` to `red`, the broker takes increasingly serious action to reclaim memory and to throttle back message producers, eventually stopping the flow of messages into the broker.

You can configure the broker's memory management functions using properties that set limits on the total number and total size of messages in memory, and that adjust the utilization thresholds at which memory resources change to a new state.

These properties are detailed in Table 2-4. (For instructions on setting these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 2-4 Message Router Properties

Property Name	Description
`imq.message.expiration.interval`	Specifies how often reclamation of expired messages occurs, in seconds. Default: `60`
`imq.system.max_count`	Specifies maximum number of messages in both memory and disk (due to swapping). Additional messages will be rejected. A value of 0 means no limit. Default: `0`
`imq.system.max_size`	Specifies maximum total size (in bytes, Kbytes, or Mbytes) of messages in both memory and disk (due to swapping). Additional messages will be rejected. A value of 0 means no limit. Default: `0`
`imq.message.max_size`	Specifies maximum allowed size (in bytes, Kbytes, or Mbytes) of a message body. Any message larger than this will be rejected. A value of 0 means no limit. Default: `70m` (Mbytes)
`imq.`*resource_state*`.threshold`	Specifies the percent memory utilization at which each memory resource state is triggered. The resource state can have the values `green`, `yellow`, `orange`, and `red`. Defaults: 0, 60, 75, and 90, respectively
`imq.redelivered.optimization`	Specifies (`true/false`) whether Message Router optimizes performance by setting Redeliver flag whenever messages are re-delivered (`true`) or only when it is logically necessary to do so (`false`). Default: `true`
`imq.transaction.autorollback`	Specifies (`true/false`) whether distributed transactions left in a `PREPARED` state are automatically rolled back when a broker is started up. If `false`, you must manually commit or roll back transactions using `imqcmd` (see "Managing Transactions" on page 157). Default: `false`

Persistence Manager

For a broker to recover, in case of failure, it needs to recreate the state of its message delivery operations. This requires it to save all persistent messages, as well as essential routing and delivery information, to a data store. A *Persistence Manager* component manages the writing and retrieval of this information.

To recover a failed broker requires more than simply restoring undelivered messages. The broker must also be able to do the following:

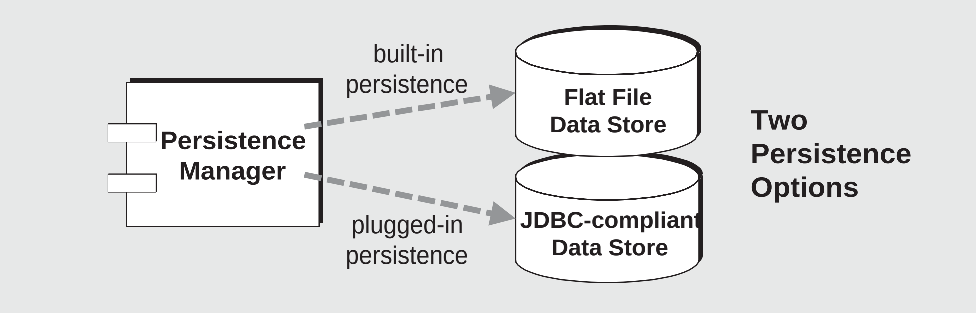
- re-create destinations
- restore the list of durable subscriptions for each topic
- restore the acknowledge list for each message
- reproduce the state of all committed transactions

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

When a broker restarts, it recreates destinations and durable subscriptions, recovers persistent messages, restores the state of all transactions, and recreates its routing table for undelivered messages. It can then resume message delivery.

MQ supports both built-in and plugged-in persistence modules (see Figure 2-4). Built-in persistence is based on a flat file data store. Plugged-in persistence uses a Java Database Connectivity (JDBC) interface and requires a JDBC-compliant data store. The built-in persistence is generally faster than plugged-in persistence; however, some users prefer the redundancy and administrative features of using a JDBC-compliant database system.

Figure 2-4 Persistence Manager Support



Built-in persistence

The default MQ persistent storage solution is a flat file store. This approach uses individual files to store persistent data, such as messages, destinations, durable subscriptions, and transactions.

The flat file data store is located at:

> `IMQ_VARHOME/instances/`*brokerName*`/filestore/`
> (`/var/imq/instances/`*brokerName*`/filestore/` on Solaris)

where *brokerName* is a name identifying the broker instance.

The file-based data store is structured so that persistent messages are each stored in their own respective file, one message per file. Destinations, durable subscriptions, and transactions, however, are all stored in a separate file for each, all destinations in one file, all durable subscriptions in another, and so on.

To create and delete files, as messages are added to and deleted from the data store, involves expensive file system operations. The MQ implementation therefore reuses these message files: when a file is no longer needed, instead of being deleted, it is added to a pool of free files available for re-use. You can configure the size of this file pool. You can also specify the percentage of free files in the file pool that are cleaned up (truncated to zero), as opposed to being simply tagged for reuse (not truncated). The higher the percentage of cleaned files, the less disk space—but the more overhead—is required to maintain the file pool. You can also specify whether or not tagged files will be cleaned up at shutdown. If the files are cleaned up, they will take up less disk space, but the broker will take longer to shut down.

The speed of storing messages in the flat file store is affected by the number of file descriptors available for use by the data store; a large number of descriptors will allow the system to process large numbers of persistent messages faster. For information on increasing the number of file descriptors, see the "Technical Notes" section of the MQ *Release Notes*.

Also, in the case of the destination file store, it is more efficient to add destinations to a fixed-size file than to increase the size of the file as destinations are added. Therefore, you can improve performance by setting the original size of the destination file in accordance with the number of destinations you expect it to ultimately store (each destination consumes about 500 bytes).

Because the data store can contain messages with proprietary information, it is recommended that the *brokerName*`/filestore/` directory be secured against unauthorized access. For instructions, see the "Technical Notes" section of the *MQ Release Notes*.

Plugged-in persistence

You can set up a broker to access any data store accessible through a JDBC driver. This involves setting a number of JDBC-related broker configuration properties and using the Database manager utility (`imqdbmgr`) to create a data store with the proper schema. The procedures and related configuration properties are detailed in Appendix A, "Setting Up Plugged-in Persistence."

Persistence-related configuration properties are detailed in Table 2-5 on page 56. (For instructions on setting these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 2-5 Persistence Properties

Property Name	Description
`imq.persist.store`	Specifies whether the broker is using built-in, file-based (`file`) persistence or plugged-in JDBC-compliant (`jdbc`) persistence. Default: `file`
`imq.persist.file.destination.file.size`	For built-in, file-based persistence, specifies the initial size of the file used to store destinations. Default: `1m` (Mbytes)
`imq.persist.file.message.filepool.limit`	For built-in, file-based persistence, specifies the maximum number of free files available for reuse in the file pool. The larger the number the faster the broker can process persistent data. Free files in excess of this value will be deleted. The broker will create and delete additional files, in excess of this limit, as needed. Default: `10000`
`imq.persist.file.message.filepool.cleanratio`	For built-in, file-based persistence, specifies the percentage of free files in the file pool that are maintained in a *clean* state (truncated to zero). The higher this value, the more overhead required to clean files during operation, but the less disk space required for the file pool. Default: `0`
`imq.persist.file.message.cleanup`	For built-in, file-based persistence, specifies whether or not the broker cleans up free files in the file store on shutdown. A value of false speeds up broker shutdown, but requires more disk space for the file store. Default: `false`

Table 2-5 Persistence Properties *(Continued)*

Property Name	Description
`imq.persist.file.message.fdpool.limit`	For built-in, file-based persistence, specifies the maximum number of data files to keep open (that is, the size of the file descriptor pool). A larger number increases the performance of persistence operations, but at the expense of other broker operations that require file descriptors, such as creating client connections. Default: `25` (Solaris and Linux), `1024` (Windows)
`imq.persist.file.sync.enabled`	Specifies whether persistence operations synchronize in-memory state with the physical storage device. If `true`, data loss due to system crash is eliminated, but at the expense of performance of persistence operations. Default: `false`

Security Manager

MQ provides authentication and authorization (access control) features, and also supports encryption capabilities.

The authentication and authorization features depend upon a user repository (see Figure 2-5 on page 58): a file, directory, or database that contains information about the users of the messaging system—their names, passwords, and group memberships. The names and passwords are used to authenticate a user when a connection to a broker is requested. The user names and group memberships are used, in conjunction with an access control file, to authorize operations such as producing or consuming messages for destinations.

MQ administrators populate an MQ-provided user repository (see "Using a Flat-File User Repository" on page 182), or plug a pre-existing LDAP user repository into the Security Manager component. The flat-file user repository is easy to use, but is also vulnerable to security attack, and should therefore be used *only* for evaluation and development purposes, while the LDAP user repository is secure and therefore best suited for production purposes.

Authentication

MQ security supports password-based authentication. When a client requests a connection to a broker, the client must submit a user name and password. The Security Manager compares the name and password submitted by the client to those stored in the user repository. On transmitting the password from client to broker, the passwords are encoded using either base 64 encoding or message digest (MD5). For more secure transmission, see "Encryption (Enterprise Edition)" on page 59. You can separately configure the type of encoding used by each connection service or set the encoding on a broker-wide basis.

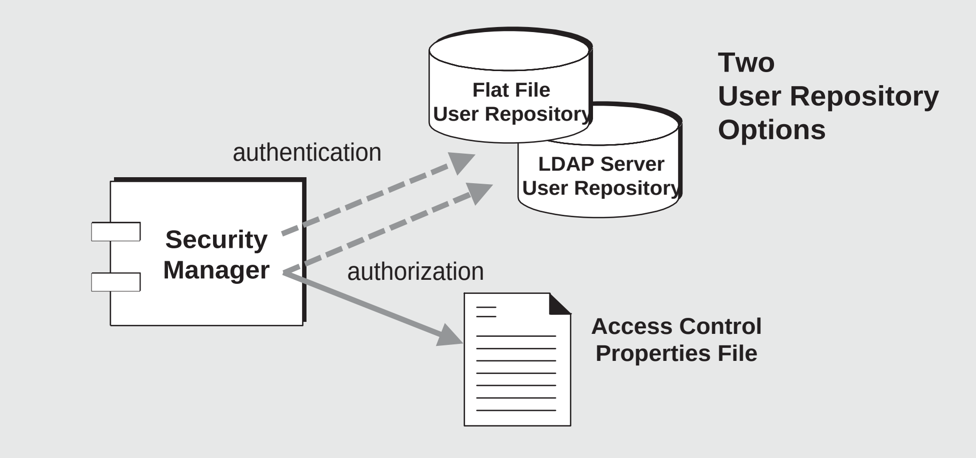
Authorization

Once the user of a client application has been authenticated, the user can be authorized to perform various MQ-related activities. The Security Manager supports both user-based and group-based access control: depending on a user's name or the groups to which the user is assigned in the user repository, that user has permission to perform certain MQ operations. You specify these access controls in an access control properties file (see Figure 2-5).

When a user attempts to perform an operation, the Security Manager 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 for the following operations:

- establishing a connection with a broker
- accessing destinations: creating a consumer, a producer, or a queue browser for any given destination or all destinations
- auto-creating destinations

Figure 2-5 Security Manager Support



For MQ 3.0.1, the default access control properties file explicitly references only one group: *admin* (see "Groups" on page 185). A user in the *admin* group has admin service connection permission. The admin service lets the user perform administrative functions such as creating destinations, and monitoring and controlling a broker. A user in any other group you define cannot, by default, get an admin service connection.

As an MQ administrator you can define groups and associate users with those groups in a user repository (though groups are not fully supported in the flat-file user repository). Then, by editing the access control properties file, you can specify access to destinations by users and groups for the purpose of producing and consuming messages, or browsing messages in queue destinations. You can make individual destinations or all destinations accessible only to specific users or groups.

In addition, if the broker is configured to allow auto-creation of destinations (see "Auto-Created (vs. Admin-Created) Destinations" on page 67), you can control for whom the broker can auto-create destinations by editing the access control properties file.

Encryption (Enterprise Edition)

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.

To use an MQ SSL-based connection service, you generate a private key/public key pair using the Key Tool utility (`imqkeytool`). This utility embeds the public key in a self-signed certificate and places it in an MQ keystore. The MQ keystore is, itself, password protected; to unlock it, you have to provide a keystore password at startup time. See "Encryption: Working With an SSL Service (Enterprise Edition)" on page 198.

Once the keystore is unlocked, a broker can pass the certificate to any client requesting a connection. The client then uses the certificate to set up an encrypted connection to the broker.

The configurable properties for authentication, authorization, encryption, and other secure communications are shown in Table 2-6. (For instructions on configuring these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 2-6 Security Properties

Property Name	**Description**
`imq.authentication.type`	Specifies whether the password should be passed in base 64 coding (`basic`) or as a MD5 digest (`digest`). Sets encoding for all connection services supported by a broker. Default: `digest`
`imq.`*service_name*`.authentication.type`	Specifies whether the password should be passed in base 64 coding (`basic`) or as a MD5 digest (`digest`). Sets encoding for named connection service, overriding any broker-wide setting. Default: inherited from the value to which `imq.authentication.type` is set.
`imq.authentication.basic.user_repository`	Specifies (for base 64 coding) the type of user repository used for authentication, either file-based (`file`) or LDAP (`ldap`). For additional LDAP properties, see Table 8-5 on page 189. Default: `file`
`imq.authentication.client.response.timeout`	Specifies the time (in seconds) the system will wait for a client to respond to an authentication request from the broker. Default: `180` (seconds)
`imq.accesscontrol.enabled`	Sets access control (`true/false`) for all connection services supported by a broker. Indicates whether system will check if an authenticated user has permission to use a connection service or to perform specific MQ operations with respect to specific destinations, as specified in the access control properties file. Default: `true`
`imq.`*service_name*`.accesscontrol.enabled`	Sets access control (`true/false`) for named connection service, overriding broker-wide setting. Indicates whether system will check if an authenticated user has permission to use the named connection service or to perform specific MQ operations with respect to specific destinations, as specified in the access control properties file. Default: inherits the setting of the property `imq.accesscontrol.enabled`

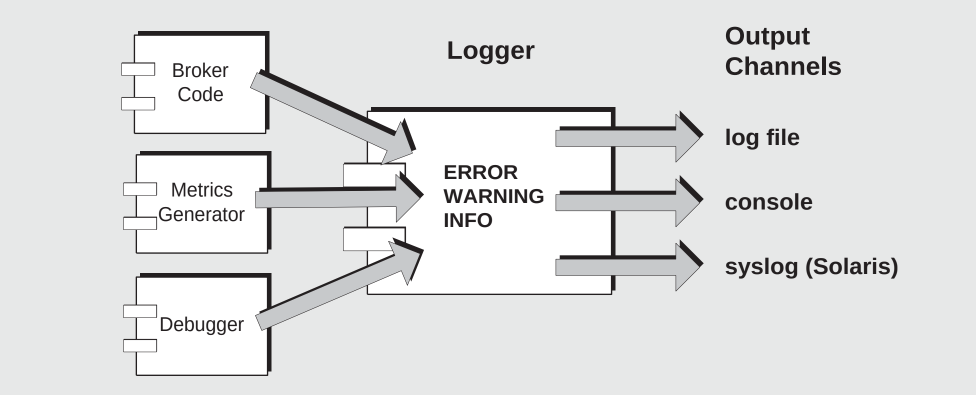
Table 2-6 Security Properties *(Continued)*

Property Name	Description
`imq.accesscontrol.file.filename`	Specifies the name of an access control properties file for all connection services supported by a broker. The file name specifies a relative file path to the directory `IMQ_HOME/etc` (`/etc/imq` on Solaris). Default: `accesscontrol.properties`
`imq.`*service_name*`.accesscontrol.file.filename`	Specifies the name of an access control properties file for named connection service. The file name specifies a relative file path to the directory `IMQ_HOME/etc` (`/etc/imq` on Solaris). Default: inherits the setting specified by `imq.accesscontrol.file.filename`.
`imq.passfile.enabled`	Specifies (true/false) if user passwords (for SSL, LDAP, JDBC) for secure communications are specified in a passfile. Default: `false`
`imq.passfile.dirpath`	Specifies the path to the directory containing the passfile. Default: `IMQ_HOME/etc (/etc/imq` on Solaris)
`imq.passfile.name`	Specifies the name of the passfile. Default: `passfile`
`imq.keystore.`*property_name*	For SSL-based services: specifies security properties relating to the SSL keystore. See Table 8-8 on page 200.

Logger

The broker includes a number of components for monitoring and diagnosing its operation. Among these are components that generate data (broker code, a metrics generator, and debugger) and a Logger component that writes out information through a number of output channels (log file, console, and Solaris syslog). The scheme is illustrated in Figure 2-6.

Figure 2-6 Logging Scheme



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

You can also specify the Logger level—ranging from the most serious and important information (errors), to less crucial information (metrics data). The categories of information, in decreasing order of criticality, are shown in Table 2-7:

Table 2-7 Logging Categories

Category	Description
`ERROR`	Messages indicating problems that could cause system failure
`WARNING`	Alerts that should be heeded but will not cause system failure
`INFO`	Reporting of metrics and other informational messages

To set the Logger level, you specify one of these categories. The logger will write out data of the specified category and all higher categories. For example, if you specify logging at the `WARNING` level, the Logger will write out warning information *and* error information.

The Logger can write data to a number of output channels: to standard output (the console), to a log file, and, on Solaris platforms, to the syslog daemon process.

For each output channel you can specify which of the categories set for the Logger will be written to that channel. For example, if the Logger level is set to `ERROR`, you can specify that you want only errors and warnings written to the console, and only info (metrics data) written to the log file. For information on configuring and using the Solaris syslog, see the syslog(1M), syslog.conf(4) and syslog(3C) man pages.

In the case of a log file, you can 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. The log file is saved at the following location:

```
IMQ_VARHOME/instances/brokerName/log/
(/var/imq/instances/brokerName/log /on Solaris)
```

An archive of the 9 most recent log files is retained as new rollover log files are created. The log files are text files that are named sequentially as follows:

```
log.txt
log_1.txt
log_2.txt
...
log_9.txt
```

The log.txt is the most recent file, and the highest numbered file is the oldest.

The configurable properties for setting the generation and logging of information by the broker are shown in Table 2-8. (For instructions on configuring these properties, see Chapter 5, “Starting and Configuring a Broker.”)

Table 2-8 Logger Properties

Property Name	**Description**
`imq.metrics.enabled`	Specifies (`true/false`) whether metrics information is being gathered. Default: `true`
`imq.metrics.interval`	Specifies the time interval, in seconds, at which metrics information is reported. A value of 0 means never. Default: `0`
`imq.log.level`	Specifies the Logger level: the categories of output that can be written to an output channel. Includes the specified category and all higher level categories as well. Values, from high to low, are: `ERROR`, `WARNING`, `INFO`. Default: `INFO`
`imq.log.file.output`	Specifies which categories of logging information are written to the log file. Allowed values are: any set of logging categories separated by vertical bars (|), or `ALL`, or `NONE`. Default: `ALL`

Table 2-8 Logger Properties *(Continued)*

Property Name	Description
`imq.log.file.dirpath`	Specifies the path to the directory containing the log file. Default: `IMQ_VARHOME/instances/`*brokerName*`/log/` (`/var/imq/instances/`*brokerName*`/log/` on Solaris)
`imq.log.file.filename`	Specifies the name of the log file. Default: `log.txt`
`imq.log.file.rolloverbytes`	Specifies the size, in bytes, of log file at which output rolls over to a new log file. A value of 0 means no rollover based on file size. Default: `0`
`imq.log.file.rolloversecs`	Specifies the age, in seconds, of log file at which output rolls over to a new log file. A value of 0 means no rollover based on age of file. Default: `604800` (one week)
`imq.log.console.output`	Specifies which categories of logging information are written to the console. Allowed values are any set of logging categories separated by vertical bars (`|`), or `ALL`, or `NONE`. Default: `ERROR| WARNING`
`imq.log.console.stream`	Specifies whether console output is written to stdout (`OUT)` or stderr (`ERR)`. Default: `ERR`
`imq.log.syslog.facility`	(Solaris only) Specifies what syslog facility the MQ broker should log as. Values mirror those listed in the syslog(3C) man page. Appropriate values for use with MQ are: `LOG_USER`, `LOG_DAEMON`, and `LOG_LOCAL0` through `LOG_LOCAL7`. Default: `LOG_DAEMON`
`imq.log.syslog.logpid`	(Solaris only) Specifies (`true/false`) whether to log the broker process ID with the message or not. Default: `true`
`imq.log.syslog.console`	(Solaris only) Specifies (`true/false`) whether to write messages to the system console if they cannot be sent to syslog. Default: `false`
`imq.log.syslog.identity`	(Solaris only) Specifies the identity string that should be prepended to every message logged to syslog. Default: `imqbrokerd_` followed by the broker instance name.

Table 2-8 Logger Properties *(Continued)*

Property Name	Description
`imq.log.syslog.output`	(Solaris only) Specifies which categories of logging information are written to syslogd(1M). Allowed values are any logging categories separated by vertical bars (`|`), or `ALL`, or `NONE`. Default: `ERROR`

Physical Destinations

MQ messaging is premised on a two-phase delivery of messages: first, delivery of a message from a producer client to a destination on the broker, and second, delivery of the message from the destination on the broker to one or more consumer clients. There are two types of destinations (see "Programming Domains" on page 37): queues (point-to-point delivery model) and topics (publish/subscribe delivery model). These destinations represent locations in a broker's physical memory where incoming messages are marshaled before being routed to consumer clients.

You create physical destinations using MQ administration tools (see "Managing Destinations" on page 152). Destinations can also be automatically created as described in "Auto-Created (vs. Admin-Created) Destinations" on page 67.

This section describes the properties and behaviors of the two types of physical destinations: queues and topics.

Queue Destinations

Queue destinations are used in point-to-point messaging, where a message is meant for ultimate delivery to only one of a number of consumers that has registered an interest in the destination. As messages arrive from producer clients, they are queued and delivered to a consumer client.

The routing of queued messages depends on the queue's delivery policy. MQ implements three queue delivery policies:

- **Single** This queue can only route messages to one message consumer. If a second message consumer attempts to register with the queue, it is rejected. If the registered message consumer disconnects, routing of messages no longer takes place and messages are saved until a new consumer is registered.

- **Failover (Enterprise Edition)** This queue can route messages to more than one message consumer, but it will only do so if its primary message consumer (the first to register with the broker) disconnects. In that case, messages will go to the next message consumer to register, and continue to be routed to that consumer until such time as that consumer fails, and so on. If no message consumer is registered, messages are saved until a consumer registers.
- **Round-Robin (Enterprise Edition)** This queue can route messages to more than one message consumer. Assuming that a number of consumers are registered for a queue, the first message into the queue will be routed to the first message consumer to have registered, the second message to the second consumer to have registered, and so on. Additional messages are routed to the same set of consumers in the same order. If a number of messages are queued up before consumers register for a queue, the messages are routed in batches to avoid flooding any one consumer. If any message consumer disconnects, the messages routed to that consumer are redistributed among the remaining active consumers. Because of such redistributions, there is no guarantee that the order of delivery of messages to consumers is the same as the order in which they are received in the queue.

Since messages can remain in a queue for an extended period of time, memory resources can become an issue. You don't want to allocate too much memory to a queue (memory is under-utilized), nor do you want to allocate too little (messages will be rejected). To allow for flexibility, based on the load demands of each queue, you can set physical properties when creating a queue: maximum number of messages in queue, maximum memory allocated for messages in queue, and maximum size of any message in queue (see Table 6-10 on page 153).

Topic Destinations

Topic destinations are used in publish/subscribe messaging, where a message is meant for ultimate delivery to all of the consumers that have registered an interest in the destination. As messages arrive from producers, they are routed to all consumers subscribed to the topic. If consumers have registered a durable subscription to the topic, they do not have to be active at the time the message is delivered to the topic—the broker will store the message until the consumer is once again active, and then deliver the message.

Messages do not normally remain in a topic destination for an extended period of time, so memory resources are not normally a big issue. However, you can configure the maximum size allowed for any message received by the destination (see Table 6-10 on page 153).

Auto-Created (vs. Admin-Created) Destinations

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

Nevertheless, there may be situations in which it is desirable for destinations to be created dynamically. For example, during a development and test cycle, you might want the broker to automatically create destinations as they are needed, without requiring the intervention of an administrator.

MQ supports this *auto-create* capability. When auto-creation is enabled, a broker automatically creates a destination whenever a MessageConsumer or MessageProducer attempts to access a non-existent destination. (The user of the client application must have auto-create privileges—see "Destination Auto-Create Access Control" on page 197).

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

You can configure an MQ message server to enable or disable the auto-create capability using the properties shown in Table 2-9. (For instructions on configuring these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 2-9 Auto-create Configuration Properties

Property Name	Description
`imq.autocreate.topic`	Specifies (`true/false`) whether a broker is allowed to auto-create a topic destination. Default: `true`
`imq.autocreate.queue`	Specifies (`true/false`) whether a broker is allowed to auto-create a queue destination. Default: `true`
`imq.queue.deliverypolicy`	Specifies the default delivery policy of auto-created queues. Values are: `single`, `round-robin`, or `failover`. Default: `single`

Temporary Destinations

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

Multi-Broker Clusters (Enterprise Edition)

The MQ Enterprise Edition supports the implementation of a message server using multiple interconnected broker instances—a broker cluster. Cluster support provides for scalability of your message server.

As the number of clients connected to a broker increases, and as the number of messages being delivered increases, a broker will eventually exceed resource limitations such as file descriptor and memory limits. One way to accommodate increasing loads is to add more brokers (that is, more broker instances) to an MQ message server, distributing client connections and message delivery across multiple brokers.

You might also use multiple brokers to optimize network bandwidth. For example, you might want to use slower, long distance network links between a set of remote brokers, while using higher speed links for connecting clients to their respective brokers.

While there are other reasons for using broker clusters (for example, to accommodate workgroups having different user repositories, or to deal with firewall restrictions), failover is *not* one of them. One broker in a cluster cannot be used as an automatic backup for another that fails. Automatic failover protection for a broker is not supported in MQ Version 3.0.1. (However, an application could be designed to use multiple brokers to implement a customized failover scheme.)

Information on configuring and managing a broker cluster is provided in "Working With Clusters (Enterprise Edition)" on page 126.

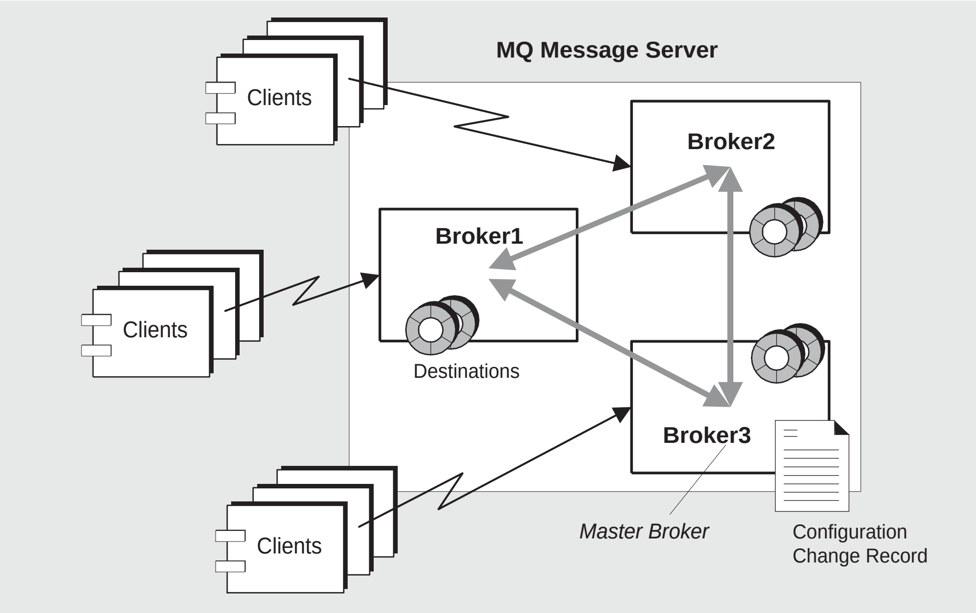
The following sections explain the architecture and internal functioning of MQ broker clusters.

Multi-Broker Architecture

A multi-broker message server allows client connections to be distributed among a number of broker instances, as shown in Figure 2-7. From a client point of view, each client connects to an individual broker (its *home* broker) and sends and receives messages as if the home broker were the only broker in the cluster. However, from a message server point of view, the home broker is working in tandem with other brokers in the cluster to provide delivery services to the message producers and consumers to which it is directly connected.

In general, the brokers within a cluster can be connected in any arbitrary topology. However, MQ Version 3.0.1 only supports fully-connected clusters, that is, a topology in which each broker is directly connected to every other broker in the cluster, as shown in Figure 2-7 on page 69.

Figure 2-7 Multi-Broker (Cluster) Architecture



In a multi-broker configuration, instances of each destination reside on all of the brokers in a cluster. In addition, each broker knows about message consumers that are registered with all other brokers. Each broker can therefore route messages from its own directly-connected message producers to remote message consumers, and deliver messages from remote producers to its own directly-connected consumers.

In a cluster configuration, the broker to which each message producer is directly connected performs the routing for messages sent to it by that producer. Hence, a persistent message is both stored and routed by the message's *home* broker.

Whenever an administrator creates or destroys a destination on a broker, this information is automatically propagated to all other brokers in a cluster. Similarly, whenever a message consumer is registered with its home broker, or whenever a consumer is disconnected from its home broker—either explicitly or because of a client or network failure, or because its home broker goes down—the relevant information about the consumer is propagated throughout the cluster. In a similar fashion, information about *durable* subscriptions is also propagated to all brokers in a cluster.

NOTE Heavy network traffic and/or large messages can clog internal cluster connections. The increased latency can sometimes cause locking protocol timeout errors. As a result, clients might get an exception when trying to create durable subscribers or queue message consumers. Normally these problems can be avoided by using a higher speed connection.

The propagation of information about destinations and message consumers to a particular broker would normally require that the broker be on line when a change is made in a shared resource. What happens if a broker is off line when such a change is made—for example, if a broker crashes and is subsequently restarted, or if a new broker is dynamically added to a cluster?

To accommodate a broker that has gone off line (or a new broker that is added), MQ maintains a record of changes made to all persistent entities in a cluster: that is, a record of all destinations and all durable subscriptions that have been created or destroyed. When a broker is dynamically added to a cluster, it first reads destination and durable subscriber information from this *configuration change record*. When it comes on line, it exchanges information about current active consumers with other brokers. With this information, the new broker is fully integrated into the cluster.

The configuration change record is managed by one of the brokers in the cluster, a broker designated as the *Master Broker*. Because the Master Broker is key to dynamically adding brokers to a cluster, you should always start this broker first. If the Master Broker is not on line, other brokers in the cluster will not be able to complete their initialization.

If a Master Broker goes off line, the configuration change record cannot be accessed by other brokers, and MQ will not allow destinations and durable subscriptions to be propagated throughout the cluster. Under these conditions, you will get an exception if you try to create or destroy destinations or durable subscriptions (or attempt a number of related operations like re-activating a durable subscription).

In a mission-critical application environment it is a good idea to make a periodic backup of the configuration change record to guard against accidental corruption of the record and safeguard against Master Broker failure. You can do this using the `-backup` option of the `imqbrokerd` command (see Table 5-2 on page 121), which provides a way to create a backup file containing the configuration change record. You can subsequently restore the configuration change record using the `-restore` option.

If necessary you can change the broker serving as the Master Broker by backing up the configuration change record, modifying the appropriate cluster configuration property (see Table 2-10 on page 72) to designate a new Master Broker, and restarting the new Master Broker using the `-restore` option.

Using Clusters in Development Environments

In development environments, where a cluster is used for testing, and where scalability and broker recovery are *not* important considerations, there is little need for a Master Broker. In environments configured *without* a Master Broker, MQ relaxes the requirement that a Master Broker be running in order to start other brokers, and allows changes in destinations and durable subscriptions to be made and to be propagated to all running brokers in a cluster. If a broker goes off line and is subsequently restored, however, it will not sync up with changes made while it was off line.

Under test situations, destinations are generally auto-created (see "Auto-Created (vs. Admin-Created) Destinations" on page 67) and durable subscriptions to these destinations are created and destroyed by the applications being tested. These changes in destinations and durable subscriptions will be propagated throughout the cluster. However, if you reconfigure the environment to use a Master Broker, MQ will re-impose the requirement that the Master Broker be running for changes to be made in destinations and durable subscriptions, and for these changes to be propagated throughout the cluster.

Cluster Configuration Properties

Each broker in a cluster must be passed information at startup time about other brokers in a cluster (host names and port numbers). This information is used to establish connections between the brokers in a cluster. Each broker must also know the host name and port number of the Master Broker (if one is used).

All brokers in a cluster should use the same cluster configuration properties. You can achieve this by placing them in one central *cluster configuration file* that is referenced by each broker at startup time.

(You can also duplicate these configuration properties and provide them to each broker individually. However, this is not recommended because it can lead to inconsistencies in the cluster configuration. Keeping just one copy of the cluster configuration properties makes sure that all brokers see the same information.)

MQ cluster configuration properties are shown in Table 2-10. (For instructions on setting these properties, see "Working With Clusters (Enterprise Edition)" on page 126.)

Table 2-10 Cluster Configuration Properties

Property Name	Description
`imq.cluster.brokerlist`	Specifies all the brokers in a cluster. Consists of a comma-separated list of *host:port* entries, where *host* is the host name of each broker and *port* is its Port Mapper port number.
`imq.cluster.masterbroker`	Specifies which broker in a cluster (if any) is the Master Broker that keeps track of state changes. Property consists of *host:port* where *host* is the host name of the Master Broker and *port* is its Port Mapper port number.
`imq.cluster.url`	Specifies the location of a cluster configuration file. Used in cases where brokers reference one central configuration file rather than being individually supplied with cluster properties values. Consists of a URL string: If kept on a web server it can be accessed using a normal `http:`*URL*. If kept on a shared drive it can be accessed using a `file:`*URL*
`imq.cluster.port`	For *each* broker within a cluster, can be used to specify the port number for the cluster connection service. The cluster connection service is used for internal communication between brokers in a cluster. Default: `0` (port is dynamically allocated)

Table 2-10 Cluster Configuration Properties *(Continued)*

Property Name	Description
`imq.cluster.hostname`	For *each* broker within a cluster, can be used to specify the host (hostname or IP address) to which the cluster connection service binds if there is more than one host available (for example, if there is more than one network interface card in a computer). The cluster connection service is used for internal communication between brokers in a cluster. Default: `null` (all available hosts)

The cluster configuration file can be used for storing all broker configuration properties that are common to a set of brokers. Though it was originally intended for configuring clusters, it can also be used to store other broker properties common to all brokers in a cluster.

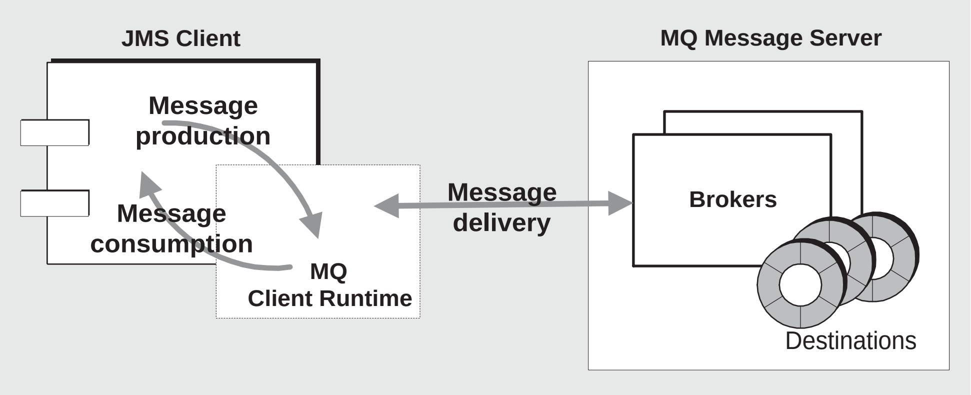
MQ Client Runtime

The MQ client runtime provides client applications with an interface to the MQ message server—it supplies client applications with all the JMS programming objects introduced in "JMS Programming Model" on page 32. It supports all operations needed for clients to send messages to destinations and to receive messages from such destinations.

This section provides a high level description of how the MQ client runtime works. Factors that affect its performance are discussed in the MQ *Developer's Guide* because they impact client application design and performance.

Figure 2-8 on page 74 illustrates how message production and consumption involve an interaction between client applications and the MQ client runtime, while message delivery involves an interaction between the MQ client runtime and the MQ message server.

Figure 2-8 Messaging Operations



Message Production

In message production, a message is created by the client, and sent over a connection to a destination on a broker. If the message delivery mode of the MessageProducer object has been set to persistent (guaranteed delivery, once and only once), the client thread blocks until the broker acknowledges that the message was delivered to its destination and stored in the broker's persistent data store. If the message is not persistent, no broker acknowledgement message (referred to as "Ack" in property names) is returned by the broker, and the client thread does not block.

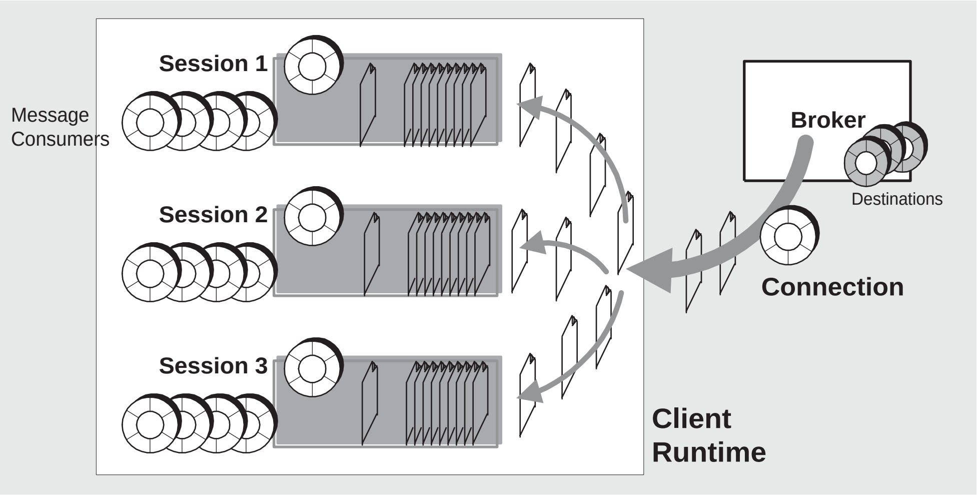
Message Consumption

Message consumption is more complex than production. Messages arriving at a destination on a broker are delivered over a connection to the MQ client runtime under the following conditions:

- the client has set up a consumer for the given destination
- the selection criteria for the consumer, if any, match that of messages arriving at the given destination
- the connection has been told to start delivery of messages.

Messages delivered over the connection are distributed to the appropriate MQ sessions where they are queued up to be consumed by the appropriate MessageConsumer objects, as shown in Figure 2-9. Messages are fetched off each session queue one at a time (a session is single threaded) and consumed either synchronously (by a client thread invoking the `receive` method) or asynchronously (by the session thread invoking the `onMessage` method of a MessageListener object).

Figure 2-9 Message Delivery to MQ Client Runtime



When a broker delivers messages to the client runtime, it marks the messages accordingly, but does not really know if they have been received or consumed. Therefore, the broker waits for the client to acknowledge receipt of a message before deleting the message from the broker's destination.

MQ Administered Objects

Administered Objects allow client application code to be provider-independent. They do this by encapsulating provider-specific implementation and configuration information in objects that are used by client applications in a provider-independent way. Administered objects are created and configured by an administrator, stored in a name service, and accessed by client applications through standard JNDI lookup code.

MQ provides two types of administered objects: ConnectionFactory and Destination. While both encapsulate provider-specific information, they have very different uses within a client application. ConnectionFactory objects are used to create connections to the message server and Destination objects are used to identify physical destinations.

Administered objects make it very easy to control and manage an MQ message server:

- You can control the behavior of connections by requiring client applications to access pre-configured ConnectionFactory objects (see "Administered Object Attributes" on page 167).
- You can control the proliferation of physical destinations by requiring client applications to access pre-configured Destination objects that correspond to existing physical destinations. (You also have to disable the brokers's auto-create capability—see "Auto-Created (vs. Admin-Created) Destinations" on page 67).
- You can control MQ message server resources by overriding message header values set by client applications (see "Administered Object Attributes" on page 167).

This arrangement therefore gives you, as an MQ administrator, control over message server configuration details, and at the same time allows client applications to be provider-independent: they do not have to know about provider-specific syntax and object naming conventions (see "JMS Provider Independence" on page 37) or provider-specific configuration properties.

You create administered objects using MQ administration tools, as described in Chapter 7, "Managing Administered Objects". When creating an administered object, you can specify that it be read only—that is, client applications are prevented from changing MQ-specific configuration values you have set when creating the object. In other words, client code cannot set attribute values on read-only administered objects, nor can you override these values using client application startup options, as described in "Overriding Attribute Values at Client Startup" on page 78.

While it is possible for client applications to instantiate both ConnectionFactory and Destination administered objects on their own, this practice undermines the basic purpose of an administered object—to allow you, as an MQ administrator, to control broker resources required by an application and to tune its performance. In addition, directly instantiating administered objects makes client applications provider-specific, rather than provider-independent.

Connection Factory Administered Objects

A `ConnectionFactory` object is used to establish physical connections between a client application and an MQ message server. It is also used to specify behaviors of the connection and of the client runtime that is using the connection to access a broker.

If you wish to support distributed transactions (see "Local Transactions" on page 40), you need to use a special `XAConnectionFactory` object that supports distributed transactions.

To create a `ConnectionFactory` administered object, see "Adding a Connection Factory" on page 174.

By configuring a `ConnectionFactory` administered object, you specify the attribute values (the properties) common to all the connections that it produces. `ConnectionFactory` and `XAConnectionFactory` objects share the same set of attributes. These attributes are grouped into a number of categories, depending on the behaviors they affect:

- Connection specification
- Auto-reconnect behavior
- Client identification
- Message header overrides
- Reliability and flow control
- Queue browser behavior
- Application server support
- JMS-defined properties support

Each of these categories and its corresponding attributes is discussed in some detail in the MQ *Developer's Guide*. While you, as an MQ administrator, might be called upon to adjust the values of these attributes, it is normally an application developer who decides which attributes need adjustment to tune the performance of client applications. Table 7-3 on page 167 presents an alphabetical summary of the attributes.

Destination Administered Objects

A `Destination` administered object represents a physical destination (a queue or a topic) in a broker to which the publicly-named `Destination` object corresponds. Its two attributes are described in Table 2-11. By creating a `Destination` object, you allow a client application's MessageConsumer and/or MessageProducer objects to access the corresponding physical destination.

To create a `Destination` administered object, see "Adding a Topic or Queue" on page 175.

Table 2-11 Destination Attributes

Attribute/property name	Description
`imqDestinationName`	Specifies the provider-specific name of the physical destination. You specify this name when you create a physical destination. Destination names must contain only alphanumeric characters (no spaces) and can begin with an alphabetic character or the characters "_" and "$". Default: `Untitled_Destination_Object`
`imqDestinationDescription`	Specifies information useful in managing the object. Default: `A Description for the Destination Object`

Overriding Attribute Values at Client Startup

As with any Java application, you can start messaging applications using the command-line to specify system properties. This mechanism can also be used to override attribute values of administered objects used in client application code. For example, you can override the configuration of an administered object accessed through a JNDI lookup in client application code.

To override administered object settings at client application startup, you use the following command line syntax:

```
java [[-Dattribute=value ]...] clientAppName
```

where *attribute* corresponds to any of the `ConnectionFactory` administered object attributes documented in "Connection Factory Administered Objects" on page 167.

For example, if you want a client application to connect to a different broker than that specified in a ConnectionFactory administered object accessed in the client code, you can start up the client application using command line overrides to set the `imqBrokerHostName` and `imqBrokerHostPort` of another broker.

If an administered object has been set as read-only, however, the values of its attributes cannot be changed using command-line overrides. Any such overrides will simply be ignored.

Chapter 3

MQ Administration

Sun™ ONE Message Queue (MQ) administration consists of a number of tasks and a number of tools for performing those tasks.

This chapter first provides an overview of administrative tasks and then describes the administration tools, focusing on common features of the command line administration utilities.

MQ Administration Tasks

The specific tasks you need to perform depend on whether you are in a development or a production environment.

Development Environments

In a development environment, the work focuses on programming MQ client applications. The MQ message server is needed principally for testing. In a development environment, the emphasis is on flexibility, and administration is minimal—consisting mostly of starting up a broker for developers to use in testing. Default implementations of the data store, user repository, access control properties file, and object store are usually adequate for developmental testing. If you are performing multi-broker testing, you probably would not use a Master Broker. In addition, the applications being tested can generally use auto-created destinations and you may not want to use centrally-managed administered objects.

Production Environments

In a production environment, in which applications must be reliably deployed and run, administration is much more important. The administration tasks you have to perform depend on the complexity of your messaging system and the complexity of the applications it must support. In general, however, these tasks can be grouped into setup operations and maintenance operations.

Setup Operations

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

- security (see Chapter 8, "Security Management"):
  - make entries into the file-based user repository or configure the broker to use an existing LDAP user repository

    (At a minimum, you want to password protect administration capability.)
  - modify access settings in the access control properties file
  - set up SSL-based connection services
- administered objects (see Chapter 7, "Managing Administered Objects"):
  - configure or set up an LDAP object store
  - create ConnectionFactory and destination administered objects
- broker clusters (see "Working With Clusters (Enterprise Edition)" on page 126):
  - create a central configuration file
  - use a Master Broker
- persistence: configure the broker to use plugged-in persistence, rather than built-in persistence (see Appendix A, "Setting Up Plugged-in Persistence")

Maintenance Operations

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

- application management:
    - disable the broker's auto-create capability (see Table 2-9 on page 67)
    - create physical destinations on behalf of applications (see "Creating Destinations" on page 153)
    - set user access to destinations (see "Authorizing Users: the Access Control Properties File" on page 192)
    - monitor and manage destinations (see "Managing Destinations" on page 152)
    - monitor and manage durable subscriptions (see "Managing Durable Subscriptions" on page 155)
    - monitor and manage transactions (see "Managing Transactions" on page 157)
- broker administration and tuning:
    - use broker metrics to tune and reconfigure the broker
    - manage broker memory resources
    - add brokers to clusters to balance loads
    - recover failed brokers
- managing administered objects
    - create additional ConnectionFactory and destination administered objects as needed
    - adjust ConnectionFactory attribute values to improve performance and throughput (see "Connection Factory Administered Objects" on page 77)

MQ Administration Tools

MQ administration tools fall into two categories: command line utilities and a graphical user interface (GUI) Administration Console (`imqadmin`). The Console combines the capabilities of two command line utilities: the Command utility (`imqcmd`) and the Object Manager utility (`imqobjmgr`). You can use the Console (and these two command line utilities) to manage a broker remotely and to manage MQ administered objects. The other command line utilities (`imqbrokerd`, `imqusermgr`, `imqdbmgr`, and `imqkeytool`) must be run on the same host as their associated broker, as shown in Figure 3-1.

Information on the Administration Console is available in the online help. The command line utilities, which are generally used to perform specialized tasks, are described in "Summary of Command Line Utilities."

The Administration Console

You can use the administration console to do the following:

- Connect to a broker and manage it.
- Create physical destinations on the broker
- Connect to an object store
- Add administered objects to the object store.

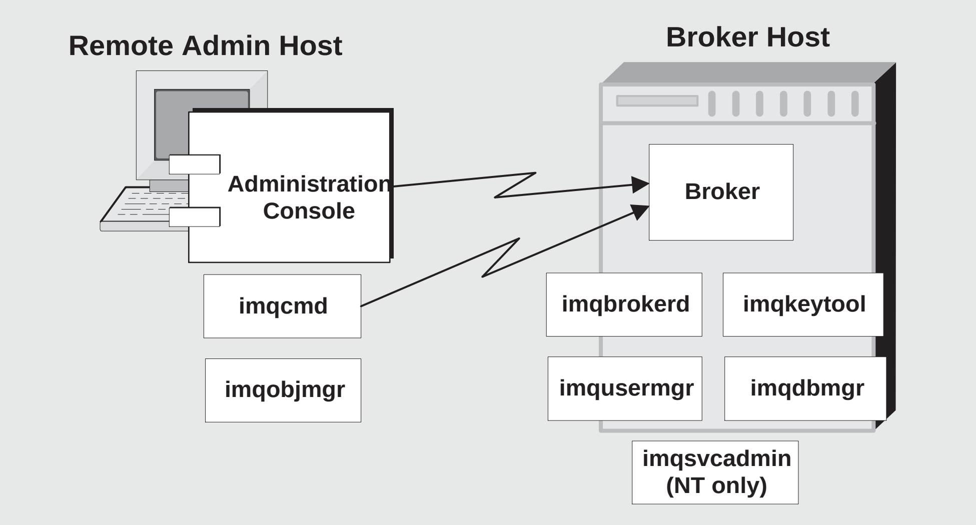
There are some tasks that you cannot use the Administration Console to perform; chief among these are starting up a broker, creating broker clusters, configuring more specialized properties of a broker, and managing a user database.

Chapter 4, "Administration Console Tutorial" provides a brief, hands-on tutorial to familiarize you with the Console and to illustrate how you use it to accomplish basic tasks.

Summary of Command Line Utilities

This section introduces the command line utilities you use to perform MQ administration tasks. You use the MQ utilities to start up and manage a broker and to perform other, more specialized administrative tasks.

Figure 3-1 Local and Remote Administration Utilities



All MQ utilities are accessible from a command line interface (CLI). Utility commands share common formats, syntax conventions, and options, as described in a subsequent section of this chapter. You can find more detailed information on the use of the command line utilities in subsequent chapters.

Broker (`imqbrokerd`) You use the Broker utility to start the broker. You use options to the `imqbrokerd` command to specify whether brokers should be connected in a cluster and to specify additional configuration information. This utility is described in Chapter 5, "Starting and Configuring a Broker."

Command (`imqcmd`) After starting a broker, you use the Command utility to create, update, and delete physical destinations; control the broker and its connection services; and manage the broker's resources.You use the `imqcmd` command to run this utility. This utility is described in Chapter 6, "Broker and Application Management."

Object Manager (`imqobjmgr`) You use the Object Manager utility to add, list, update, and delete administered objects in an object store accessible via JNDI. Administered objects allow JMS clients to be provider-independent by insulating them from JMS provider-specific naming and configuration formats. You use the `imqobjmgr` command to run this utility. This utility is described in Chapter 7, "Managing Administered Objects."

User Manager (`imqusermgr`) You use the User Manager utility to populate a file-based user repository used to authenticate and authorize users. You use the `imqusermgr` command to run this utility. This utility is described in Chapter 8, "Security Management."

Key Tool (`imqkeytool`) You use the Key Tool utility to generate self-signed certificates used for SSL authentication. You use the `imqkeytool` command to run this utility, which is described in Chapter 8, "Security Management" and in Appendix B, "HTTP/HTTPS Support (Enterprise Edition)."

Database Manager (`imqdbmgr`) You use the Database Manager utility to create and manage a JDBC-compliant database used for persistent storage. You use the `imqdbmgr` command to run this utility. For more information, see Appendix A, "Setting Up Plugged-in Persistence."

Service Administrator (`imqsvcadmin`) You use the Service Administrator utility to install, query, and remove the broker as a Windows service. For more information, see Appendix C, "Using a Broker as a Windows Service."

Command Line Syntax

MQ command-line interface utilities are simple shell commands. That is, from the standpoint of the Windows, Linux, or Solaris command shell where they are entered, the name of the utility itself is a command and its subcommands or options are simply arguments passed to that command. For this reason, there are no commands to start or quit the utility, per se, and no need for such commands.

All the command line utilities share the following command syntax:

```
Utility_Name [subcommand] [argument] [[-option_name [-option_argument]]…]
```

Utility_Name specifies the name of an MQ utility, for example, `imqcmd`, `imqobjmgr`, `imqusermgr`, and so on.

There are four important things to remember:

- Specify options *after* subcommands (and arguments, if the utility accepts both types of operands).
- If an argument contains a space, enclose the whole argument in quotation marks. It is generally safest to enclose an attribute-value pair in quotes.
- If you specify the -v (version) or the -h/-H (help) options on a command line, nothing else on that command line is executed. See Table 3-1 on page 87 for a description of common options.
- Separate the subcommand, arguments, options, and option arguments with spaces.

The following is an example of a command line that has no subcommand clause. The command starts the default broker.

```
imqbrokerd
```

The following command is a bit more complicated: it destroys a destination of type `queue` that is named `myQueue` for an administrator (user) named `admin` with a corresponding password `admin`, without confirmation and without output being displayed on the console.

```
imqcmd destroy dst -t q -n myQueue -u admin -p admin -f -s
```

Common Command Line Options

Table 3-1 describes the options that are common to all MQ administration utilities. Aside from the requirement that you specify these options *after* you specify the subcommand on the command line, the options described below (or any other options passed to a utility) do not have to be entered in any special order.

Table 3-1 Common MQ Command Line Options

Option	Description
`-h`	Displays usage help for the specified utility.
`-H`	Displays expanded usage help, including attribute list and examples (supported only for `imqcmd` and `imqobjmgr`).
`-s`	Turns on silent mode: no output is displayed. Specify as `-silent` for `imqbrokerd`.
`-v`	Displays version information.
`-f`	Performs the given action without prompting for user confirmation.
`-pre`	(Used only with `imqobjmgr`) Turns on preview mode, allowing the user to see the effect of the rest of the command line without actually performing the command. This can be useful in checking for the value of default attributes.
`-javahome` *path*	Specifies the location of an alternate Java 2 runtime to use.

Chapter 4

Administration Console Tutorial

This tutorial focuses on the use of the Administration Console, a graphical interface for administering a Sun™ ONE Message Queue (MQ) message server. By following this tutorial, you will learn how to do the following:

- Start a broker and use the Console to connect to it and manage it
- Create physical destinations on the broker
- Create an object store and use the Console to connect to it
- Add administered objects to the object store

The tutorial is designed to set up the destinations and administered objects needed to run a simple JMS-compliant application, `SimpleJNDIClient`. In the last part of the tutorial you run this application.

This tutorial is provided mainly to guide you through performing basic administration tasks using the Administration Console. It is not a substitute for reading through the MQ *Developer's Guide* or other chapters of this *Administrator's Guide.*

Some MQ administration tasks cannot be accomplished using graphical tools; you will need to use command line utilities to perform such tasks as the following:

- Configuring certain broker properties

  Some broker properties cannot be configured using the Administration Console. These can be configured as described in Chapter 5, "Starting and Configuring a Broker" on page 113 or in "Updating a Broker" on page 147.

- Creating broker clusters

  See“Working With Clusters (Enterprise Edition)” on page 126 for more information.

- Managing a user database

  See “Authenticating Users” on page 182 for more information.

Getting Ready

Before you can start this tutorial you must install the MQ product. For more information, see the MQ *Installation Guide*. Note that this tutorial is Windows-centric, with added notes for unix users.

In this tutorial, choosing Item1 > Item2 > Item3 means that you should pull down the menu called Item1, choose Item2 from that menu and then choose Item3 from the selections offered by Item2.

Starting the Administration Console

The Administration Console is a graphical tool that you use to do the following:

- Create references to and connect to brokers
- Administer brokers
- Create physical destinations on the brokers, which are used by the broker for message delivery
- Connect to object stores in which you place MQ administered objects

  Administered objects allow you to manage the messaging needs of JMS-compliant applications. For more information, see “MQ Administered Objects” on page 75.

➤ To start the Administration Console

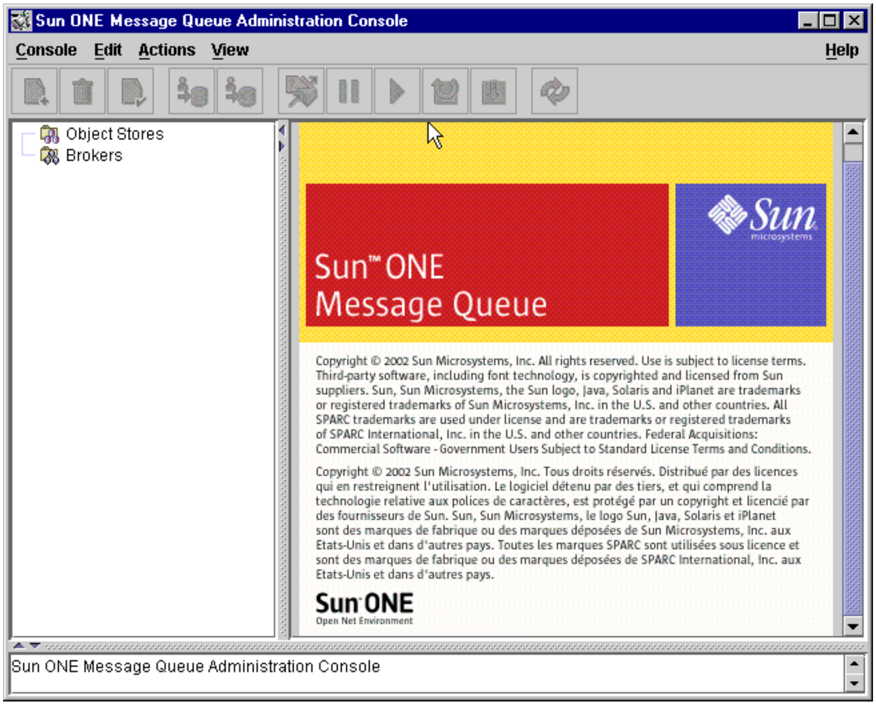
1. Choose Start > Programs > Sun ONE Message Queue 3.0> Administration.

   You may need to wait a few seconds before the Console window is displayed.

   Non-Windows users: enter the following command at the command prompt:

   ```
   $IMQ_HOME/bin/imqadmin (/usr/bin/imqadmin on Solaris)
   ```

2. Take a few seconds to examine the Console window.



Copyright © 2002 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers. Sun, Sun Microsystems, the Sun logo, Java, Solaris and iPlanet are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Federal Acquisitions: Commercial Software - Government Users Subject to Standard License Terms and Conditions.

Copyright © 2002 Sun Microsystems, Inc. Tous droits réservés. Distribué par des licences qui en restreignent l'utilisation. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun. Sun, Sun Microsystems, le logo Sun, Java, Solaris et iPlanet sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays.

Sun ONE Message Queue Administration Console

The Console features a menu at the top, a tool bar just underneath the menu, a navigation pane to the left, a larger pane to the right (now displaying graphics identifying the Sun ONE Message Queue product), and a status pane at the bottom.

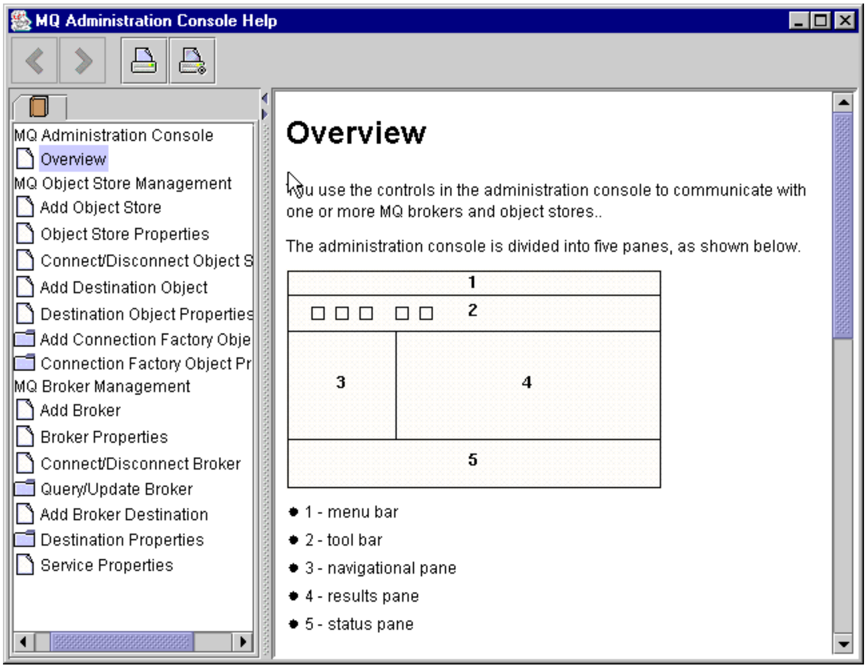
No tutorial can provide complete information, so let's first find out how to get help information for the Administration Console.

Getting Help

Locate the Help menu at the extreme right of the menu bar.

➤ To display Administration Console help information

1. Pull down the Help menu and choose Overview. A help window is displayed.



MQ Administration Console
Overview
MQ Object Store Management
Add Object Store
Object Store Properties
Connect/Disconnect Object S
Add Destination Object
Destination Object Properties
Add Connection Factory Obje
Connection Factory Object Pr
MQ Broker Management
Add Broker
Broker Properties
Connect/Disconnect Broker
Query/Update Broker
Add Broker Destination
Destination Properties
Service Properties

Overview

You use the controls in the administration console to communicate with one or more MQ brokers and object stores..

The administration console is divided into five panes, as shown below.

- 1 - menu bar
- 2 - tool bar
- 3 - navigational pane
- 4 - results pane
- 5 - status pane

Notice how the help information is organized. The left pane shows a table of contents; the right pane shows the contents of any item you select on the left.

Look at the right pane of the Help window. It shows a skeletal view of the Administration Console, identifying the use of each of the Console's panes.

2. Look at the Help window's contents pane. It organizes topics in three areas: overview, object store management, and broker management. Each of these areas contains files and folders. Each folder provides help for dialogs containing multiple tabs; each file provides help for a simple dialog or tab.

   Your first Console administration task, "Adding a Broker" on page 95, will be to create a reference to a broker you manage through the Console. Before you start, however, check the online help for information.

3. Click the Add Broker item in the Help window's contents pane.

   Note that the contents pane has changed. It now contains text that explains what it means to add a broker and that describes the use of each field in the Add Broker dialog. Field names are shown in bold text.

4. Read through the help text.

5. Close the Help window.

Working With Brokers

A broker provides delivery services for an MQ messaging system. Message delivery is a two-phase process: the message is first delivered to a physical destination on a broker and then it is delivered to one or more consuming clients.

Working with brokers involves the following tasks:

- Start and configure the broker

  You can start the broker from the Start > Programs menu on Windows or by using the `imqbrokerd` command. If you use the `imqbrokerd` command, you can specify broker configuration information using command line options. If you use the Programs menu, you can specify configuration information using the Console and in other ways described in Chapter 5, "Starting and Configuring a Broker."

  NOTE You cannot start a broker instance using the Administration Console .

- Manage the broker and its services either by using the Administration Console or by using the Command command-line utility (`imqcmd`).

- Create the physical destinations needed by client applications

- Monitor resource use to improve throughput and reliability

The broker supports communication with both application clients and administration clients. It does this by means of different connection services, and you can configure the broker to run any or all of these services. For more information about connection services, see "Connection Services" on page 46.

Starting a Broker

You cannot start a broker using the Administration Console. Start the broker as described below (also, see Chapter 5, “Starting and Configuring a Broker”).

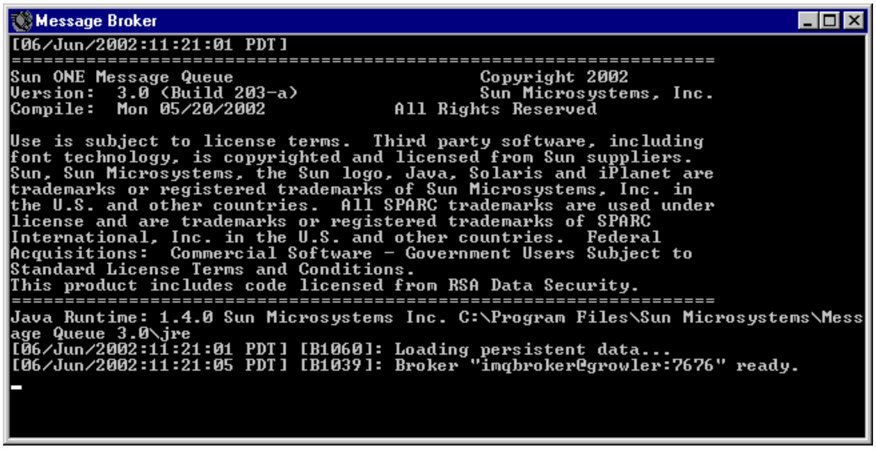
➤ To start a broker

1. Choose Start > Programs >Sun ONE Message Queue 3.0 > Broker.

   Non-Windows: enter the following command to start a broker.

   `%$IMQ_HOME/bin/imqbrokerd (/usr/bin/imqbrokerd` on Solaris)

   A broker process window is displayed. The name of the broker is specified as is the fact that it is ready.



```
[06/Jun/2002:11:21:01 PDT]
==================================================================
Sun ONE Message Queue                          Copyright 2002
Version:  3.0 (Build 203-a)                    Sun Microsystems, Inc.
Compile:  Mon 05/20/2002               All Rights Reserved

Use is subject to license terms.  Third party software, including
font technology, is copyrighted and licensed from Sun suppliers.
Sun, Sun Microsystems, the Sun logo, Java, Solaris and iPlanet are
trademarks or registered trademarks of Sun Microsystems, Inc. in
the U.S. and other countries.  All SPARC trademarks are used under
license and are trademarks or registered trademarks of SPARC
International, Inc. in the U.S. and other countries.  Federal
Acquisitions:  Commercial Software - Government Users Subject to
Standard License Terms and Conditions.
This product includes code licensed from RSA Data Security.
==================================================================
Java Runtime: 1.4.0 Sun Microsystems Inc. C:\Program Files\Sun Microsystems\Mess
age Queue 3.0\jre
[06/Jun/2002:11:21:01 PDT] [B1060]: Loading persistent data...
[06/Jun/2002:11:21:05 PDT] [B1039]: Broker "imqbroker@growler:7676" ready.
_
```

2. Bring the Administration Console window back into focus. You are now ready to add the broker to the Console and to connect to it.

You do not have to start the broker before you add a reference to it in the Administration Console, but you must start the broker before you can connect to it.

Adding a Broker

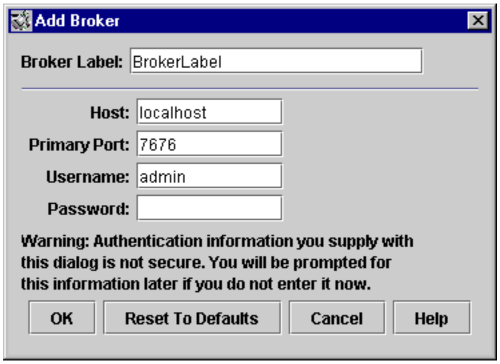
Adding a broker creates a reference to that broker in the Administration Console. After adding the broker, you can connect to it.

➤ To add a broker to the Administration Console

1. Right-click on Brokers in the navigation pane and choose Add Broker.

2. Enter `MyBroker` in the Broker Label field.

   This provides a label that identifies the broker in the Administration Console.



Add Broker

Broker Label: BrokerLabel

Host: localhost

Primary Port: 7676

Username: admin

Password:

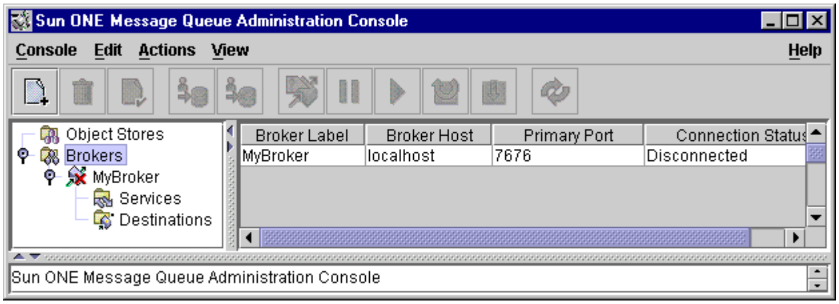
Warning: Authentication information you supply with this dialog is not secure. You will be prompted for this information later if you do not enter it now.

   Note the default host name (`localhost`) and primary port (`7676`) specified in the dialog. These are the values you will need to specify later, when you configure the connection factory that the client will use to set up connections to this broker.

   Leave the Password field blank. Your password will be more secure if you specify it at connection time.

3. Click OK to add the broker.

Look at the navigation pane. The broker you just added should be listed there under Brokers. The red X over the broker icon tells you that the broker is not currently connected to the Console.



Broker Label	Broker Host	Primary Port	Connection Status
MyBroker	localhost	7676	Disconnected

Sun ONE Message Queue Administration Console

4. Right-click on MyBroker and choose Properties from the popup menu.

   The broker properties dialog is displayed. You can use this dialog to update any of the properties you specified when you added the broker.

Changing the Administrator Password

When you connect to the broker, you are prompted for a password if you have not specified one when you added the broker. For improved security, it's a good idea to change the default administrator password (`admin`) before you connect.

➤ **To change the administrator password**

1. Open a command-prompt window or, if one is already opened, bring it forward.
2. Enter a command like the following, substituting your own password for `abracadabra`. The password you specify then replaces the default password of `admin`.

   ```
   imqusermgr update -u admin -p abracadabra
   ```

   (On Solaris and Linux, you must be root to perform this operation.)

The change takes effect immediately. You must then specify the new password whenever you use one of the MQ command line utilities or the Administration Console.

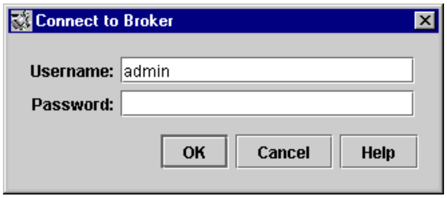
Although clients use a different connection service than administrators, they are also assigned a default user name and password so that you can test MQ without having to do extensive administrative set up. By default, a client can connect to the broker as user `guest` with the password `guest`. You should, however, establish secure user names and passwords for clients as soon as you can. See "Authenticating Users" on page 182 for more information.

Connecting to the Broker

➤ **To connect to the broker**

1. Right-click `MyBroker` and choose Connect to Broker.

   A dialog is displayed that allows you to specify your name and password.



Connect to Broker

Username: admin

Password:

2. Enter `admin` in the Password field or whatever value you specified for the password in "Changing the Administrator Password" on page 96.

   Specifying the user name `admin` and supplying the correct password connects you to the broker, with administrative privileges.

3. Click OK to connect to the broker.

After you connect to the broker, you can choose from the Actions menu to get information about the broker, to pause and resume the broker, to shutdown and restart the broker, and to disconnect from the broker.

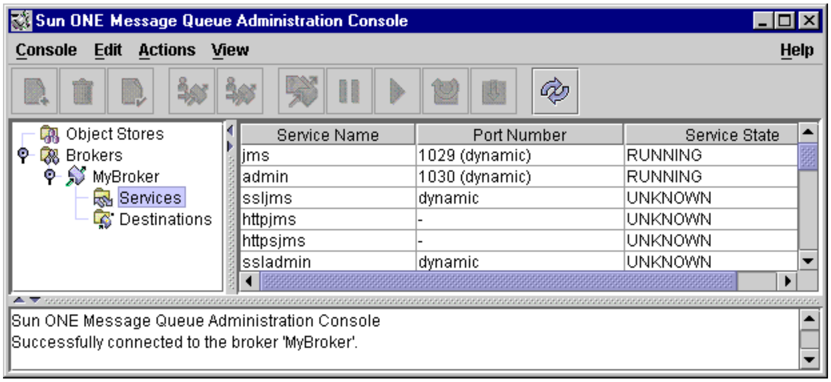
Viewing Connection Services

A broker is distinguished by the connection services it provides and the physical destinations it supports.

➤ **To view available connection services**

1. Select Services in the navigation pane.

   Available services are listed in the results pane. For each service, its name, port number, and state is provided.



Service Name	Port Number	Service State
jms	1029 (dynamic)	RUNNING
admin	1030 (dynamic)	RUNNING
ssljms	dynamic	UNKNOWN
httpjms	-	UNKNOWN
httpsjms	-	UNKNOWN
ssladmin	dynamic	UNKNOWN

```
Sun ONE Message Queue Administration Console
Successfully connected to the broker 'MyBroker'.
```

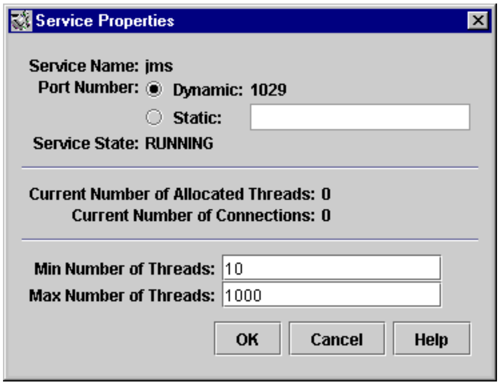
2. Select the jms service by clicking on it in the results pane.

3. Pull down the Actions menu and note the highlighted items.

   You have the option of pausing the jms service or of viewing and updating its properties.

4. Choose Properties from the Actions menu.

   Note that by using the Service Properties dialog, you can assign the service a static port number and you can change the minimum and maximum number of threads allocated for this service.



Service Name: jms
Port Number: ● Dynamic: 1029
○ Static:
Service State: RUNNING

Current Number of Allocated Threads: 0
Current Number of Connections: 0

Min Number of Threads: 10
Max Number of Threads: 1000

5. Click OK or Cancel to close the Properties dialog.
6. Select the admin service in the results pane.
7. Pull down the Actions menu.

   Notice that you cannot pause this service (the pause item is disabled). The admin service is the administrator's link to the broker. If you paused it, you would no longer be able to access the broker.
8. Choose Actions > Properties to view the properties of the admin service.
9. Click OK or Cancel when you're done.

Adding Physical Destinations to a Broker

You must explicitly create physical destinations on the broker so that JMS-compliant applications can run properly. You do not need to do this if the broker has destination auto-creation enabled, which allows it to create physical destinations dynamically.

Destination auto-creation is acceptable in a development environment. However, in a production setting, it is advisable to turn it off and have the broker use physical destinations that you have explicitly created. This allows you, the administrator, to be fully aware of the destinations that are in use on the broker.

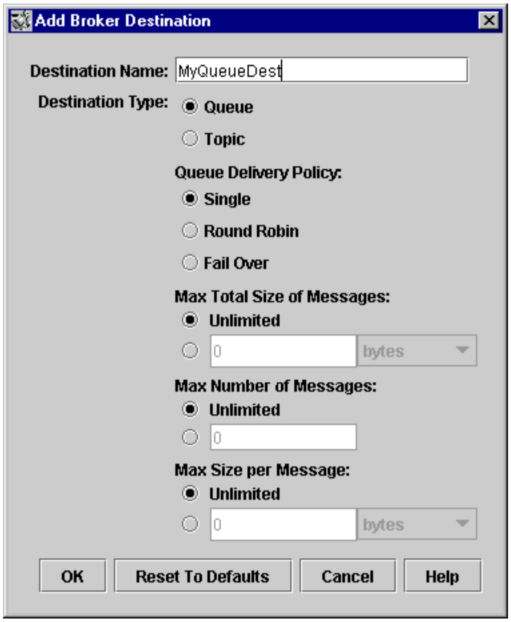
You control whether the broker can add auto-created destinations by setting the `imq.autocreate.topic` or `imq.autocreate.queue` properties. For more information, see "Auto-Created (vs. Admin-Created) Destinations" on page 67.

In this section of the tutorial, you will add a physical destination to the broker. You should note the name you assign to the destination; you will need it later when you create an administered object that corresponds to this physical destination.

➤ To add a queue destination to a broker

1. Right-click the Destinations node of MyBroker and choose Add Broker Destination.

   The following dialog is displayed:



Add Broker Destination

Destination Name: MyQueueDest

Destination Type:
● Queue
○ Topic

Queue Delivery Policy:
● Single
○ Round Robin
○ Fail Over

Max Total Size of Messages:
● Unlimited
○ 0 bytes

Max Number of Messages:
● Unlimited
○ 0

Max Size per Message:
● Unlimited
○ 0 bytes

2. Enter `MyQueueDest` in the Destination Name field.

3. Select the Queue radio button if it is not already selected.

4. Make sure the Queue Delivery Policy is selected as Single.

5. Click OK to add the physical destination.

   The destination now appears in the results pane.

Working With Physical Destinations

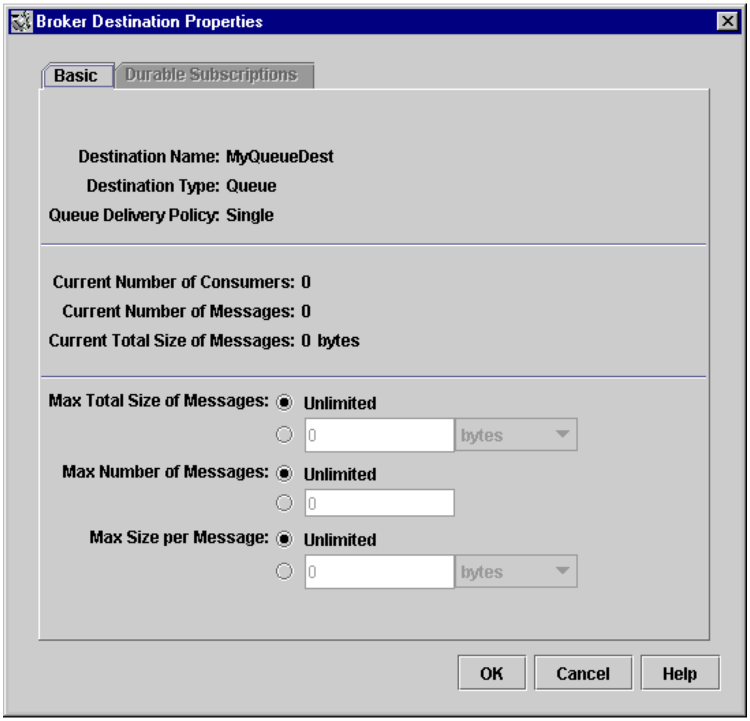
Once you have added a physical destination on the broker, you can do any of the following tasks, as described in the procedures below:

- View and update the properties of a physical destination
- Purge messages at a destination
- Delete a destination

➤ To view the properties of a physical destination

1. Select the Destinations node of MyBroker.
2. Select MyQueueDest in the results pane.
3. Choose Actions > Properties.

   The following dialog is displayed:



Broker Destination Properties

Destination Name: MyQueueDest
Destination Type: Queue
Queue Delivery Policy: Single

Current Number of Consumers: 0
Current Number of Messages: 0
Current Total Size of Messages: 0 bytes

Max Total Size of Messages: ● Unlimited ○ 0 bytes
Max Number of Messages: ● Unlimited ○ 0
Max Size per Message: ● Unlimited ○ 0 bytes

Note that the only properties you can change for a queue have to do with the size and number of messages that are delivered to that queue.

4. Click Cancel to close the dialog.

➤ To purge messages from a destination

1. Select the physical destination in the Results pane.
2. Choose Actions > Purge Messages.

   A confirmation dialog is displayed.

Purging messages removes the messages and leaves an empty destination.

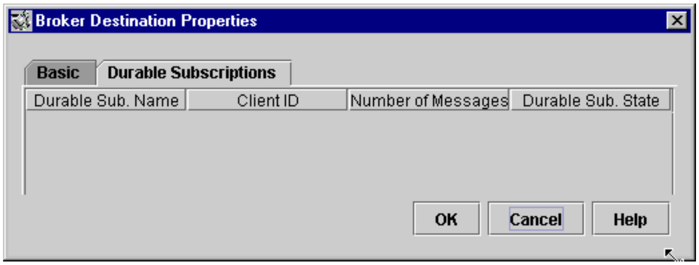
➤ To delete a destination

1. Select the physical destination in the results pane.
2. Choose Edit > Delete.

Deleting a destination purges the messages at that destination and removes the destination.

Getting Information About Topic Destinations

The dialog about topic destinations includes an additional tab that lists information about durable subscriptions.



Durable Sub. Name	Client ID	Number of Messages	Durable Sub. State

You can use this dialog to:

- purge durable subscriptions, removing all messages associated with a durable subscription
- delete durable subscriptions, purging all messages associated with a durable subscription and also removing the durable subscription

Working with Object Stores

An object store, be it an LDAP directory server or a file system store (directory in the file system), is used to store MQ administered objects that encapsulate MQ-specific implementation and configuration information about objects that are used by client applications.

Although administered objects can be instantiated and configured within client code, it is preferable that you, the administrator, create and configure these objects and store them in an object store that is accessed by client applications through standard JNDI lookup code. This allows client code to be provider-independent.

For more information about administered objects, see "MQ Administered Objects" on page 75.

You cannot use the Administration Console to *create* an object store. You must do this ahead of time as described in the following section.

Adding an Object Store

Adding an object store creates a reference to an existing object store in the Administration Console. This reference is retained even if you quit and restart the Console.

➤ **To add a file-system object store**

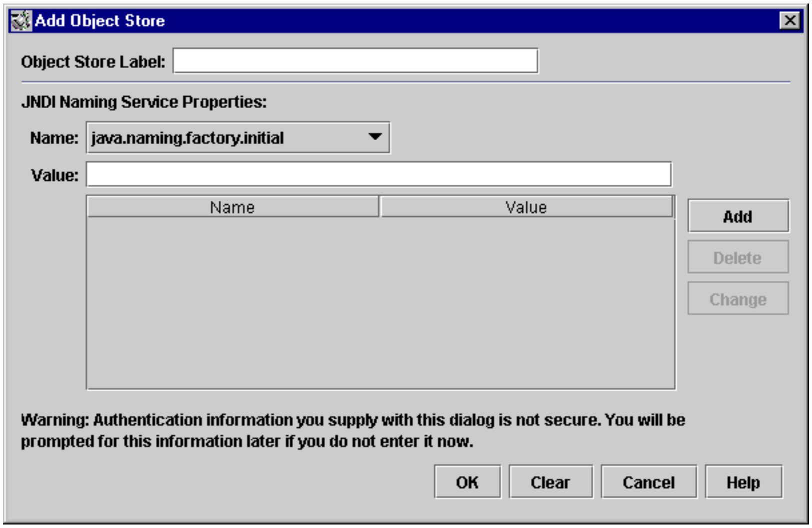
1. If you do not already have a folder named `Temp` on your C drive, create it now.

   The sample application used in this tutorial assumes that the object store is a folder named `Temp` on the C drive. In general, a file-system object store can be any directory on any drive.

   Non-Windows: you can use the `/tmp` directory, which should already exist.

2. Right-click on Object Stores and choose Add Object Store.

   The following dialog is displayed:



Add Object Store

Object Store Label:

JNDI Naming Service Properties:

Name: java.naming.factory.initial

Value:

Name	Value

Warning: Authentication information you supply with this dialog is not secure. You will be prompted for this information later if you do not enter it now.

3. Enter `MyObjectStore` in the field named ObjectStoreLabel.

   This simply provides a label for the display of the object store in the Administration Console.

   In the following steps, you will need to enter JNDI name/value pairs. These pairs are used by JMS-compliant applications for looking up administered objects.

4. From the Name pull-down menu, choose `java.naming.factory.initial`.

   This property allows you to specify what JNDI service provider you wish to use. For example, a file system service provider or an LDAP service provider.

5. In the Value field, enter the following

   ```
   com.sun.jndi.fscontext.RefFSContextFactory
   ```

   This means that you will be using a file system store. (For an LDAP store, you would specify `com.sun.jndi.ldap.LdapCtxFactory`.)

   In a production environment, you will probably want to use an LDAP directory server as an object store. For information about setting up the server and doing JNDI lookups, see "Object Store Attributes" on page 169.

6. Click the Add button.

   Notice that the property and its value are now listed in the property summary pane.

7. From the Name pull down menu, choose `java.naming.provider.url`.

   This property allows you to specify the exact location of the object store. For a file system type object store, this will be the name of an existing directory.

8. In the Value field, enter the following

   ```
   file:///C:/Temp
   ```

   (`file:///tmp` on Solaris and Linux)

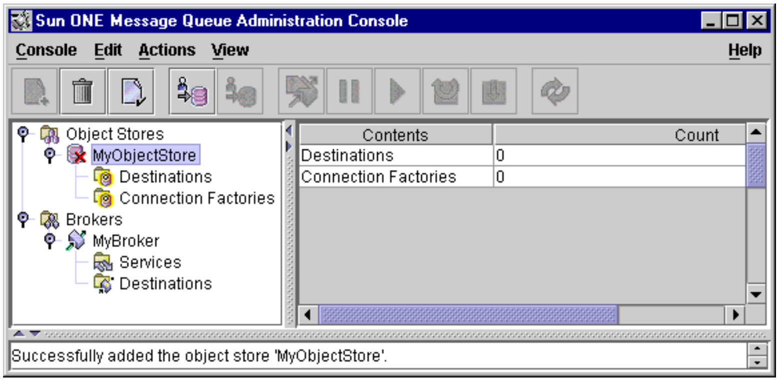
9. Click the Add button.

   Notice that both properties and their values are now listed in the property summary pane. If you were using an LDAP server, you might also have to specify authentication information; this is not necessary for a file-system store.

10. Click OK to add the object store.

11. If the node MyObjectStore is not selected in the navigation pane, select it now.

    The Administration Console now looks like this:



Contents	Count
Destinations	0
Connection Factories	0

Successfully added the object store 'MyObjectStore'.

The object store is listed in the navigation pane and its contents, Destinations and Connection Factories, are listed in the results pane. We have not yet added any administered objects to the object store, and this is shown in the Count column of the results pane.

A red X is drawn through the object store's icon in the navigation pane. This means that it is disconnected. Before you can use the object store, you will need to connect to it.

Checking Object Store Properties

While the Administration Console is disconnected from an object store, you can examine and change some of the properties of the object store.

➤ **To display the properties of an object store**

1. Right click on MyObjectStore in the navigational pane.
2. Choose Properties from the popup menu.

   A dialog is displayed that shows all the properties you specified when you added the object store. You can change any of these properties and click OK to update the old information.
3. Click OK or Cancel to dismiss the dialog.

Connecting to an Object Store

Before you can add objects to an object store, you must connect to it.

➤ **To connect to an object store**

1. Right click on MyObjectStore in the navigational pane.
2. Choose Connect to Object Store from the popup menu.

   Notice that the object store's icon is no longer crossed out. You can now add objects, connection factories and destinations, to the object store.

Adding a Connection Factory Administered Object

You can use the administration console to create and configure a connection factory. A connection factory is used by client code to connect to the broker. By configuring a connection factory, you can control the behavior of the connections it is used to create.

For information on configuring connection factories, see the online help and the MQ *Developer's Guide*.

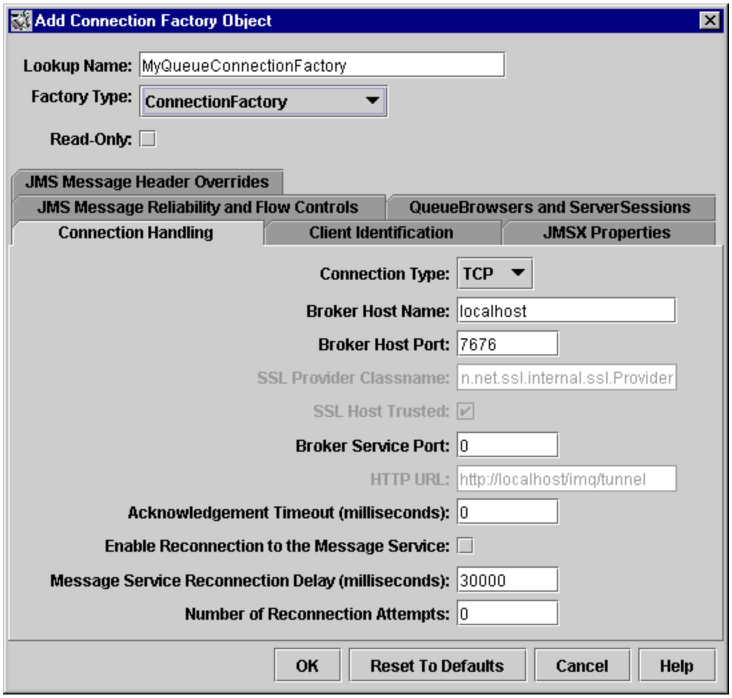
NOTE The Administration Console lists and displays only MQ administered objects. If an object store should contain a non-MQ object with the same lookup name as an administered object that you wish to add, you will receive an error when you attempt the add operation.

➤ To add a connection factory to an object store

1. If not already connected, connect to MyObjectStore (see "Connecting to an Object Store" on page 106)

2. Right click on the Connection Factories node and choose Add Connection Factory Object.

   The Add Connection Factory Object dialog is displayed.


Add Connection Factory Object
JMS Message Header Overrides
JMS Message Reliability and Flow Controls
QueueBrowsers and ServerSessions
Connection Handling
Client Identification
JMSX Properties
OK
Reset To Defaults
Cancel
Help

Lookup Name: MyQueueConnectionFactory

Factory Type: ConnectionFactory

Read-Only: □

Connection Type: TCP

Broker Host Name: localhost

Broker Host Port: 7676

SSL Provider Classname: n.net.ssl.internal.ssl.Provider

SSL Host Trusted: ☑

Broker Service Port: 0

HTTP URL: http://localhost/imq/tunnel

Acknowledgement Timeout (milliseconds): 0

Enable Reconnection to the Message Service: □

Message Service Reconnection Delay (milliseconds): 30000

Number of Reconnection Attempts: 0

3. Enter the name "MyQueueConnectionFactory" in the LookupName field.

   This is the name that the client code uses when it looks up the connection factory as shown in the following line from SimpleAdmin.java:

   ```
   qcf=(javax.jms.QueueConnectionFactory)
                       ctx.lookup("MyQueueConnectionFactory")
   ```

4. Select the QueueConnectionFactory from the pull-down menu to specify the type of the connection factory.

5. Enter the host name and port for the broker to which the client is planning to connect, in the Broker Host Name and Broker Host Port fields.

   In this tutorial, the client connects to the default broker--that is, a broker on localhost at port 7676, so you do not have to change these fields.

6. Click through the tabs for this dialog to see the kind of information that you can configure for the connection factory. Use the Help button in the lower right hand corner of the Add Connection Factory Object dialog to get information about individual tabs. Do not change any of the default values for now.

7. Click OK to create the queue connection factory.

8. Look at the results pane: the lookup name and type of the newly created connection factory are listed.

Adding a Destination Administered Object

Destination administered objects are associated with physical destinations on the broker; they point to those destinations, as it were, allowing clients to look up and find physical destinations, independently of the provider-specific ways in which those destinations are named and configured.

When a JMS client sends a message, it looks up (or instantiates) a destination administered object and references it in the send() method of the JMS API. The broker is then responsible for delivering the message to the physical destination that is associated with that administered object:

- If you have created a physical destination that is associated with that administered object, the broker delivers the message to that physical destination.

- If you have not created a physical destination and the autocreation of physical destinations is enabled, the broker itself creates the physical destination and delivers the message to that destination.

- If you have not created a physical destination and the autocreation of physical destinations is *disabled*, the broker cannot create a physical destination and cannot deliver the message.

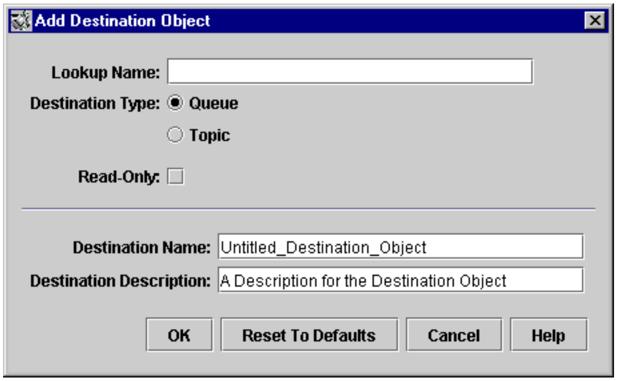
In the next part of the tutorial, you will be adding an administered object that corresponds to the physical destination you added earlier.

➤ To add a destination to an object store

1. Right-click on the Destinations node in the navigation pane.

2. Choose Add Destination Object.

   The Administration Console displays an Add Destination Object dialog that you use to specify information about the object.



Add Destination Object

Lookup Name:

Destination Type: ● Queue

○ Topic

Read-Only: □

Destination Name: Untitled_Destination_Object

Destination Description: A Description for the Destination Object

3. Enter “`MyQueue`” in the Lookup Name field.

   The lookup name is used to find the object using JNDI lookup calls. In the sample application, the call is the following:

   ```
   queue=(javax.jms.Queue)ctx.lookup(“MyQueue”);
   ```

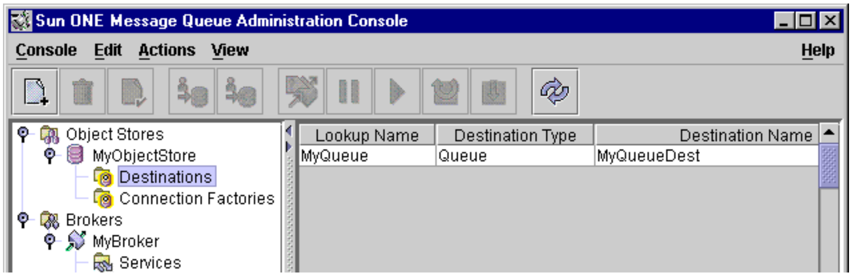
4. Select the Queue radio button for the Destination Type.

5. Enter `MyQueueDest` in the Destination Name field.

   This is the name you specified when you added a physical destination on the broker.

6. Click OK.

7. Select Destinations in the navigation pane and notice how information about the queue destination administered object you have just added is displayed in the results pane.



Lookup Name	Destination Type	Destination Name
MyQueue	Queue	MyQueueDest

Administered Object Properties

To view or update the properties of an administered object, you need to select Destinations or Connection Factories in the navigation pane, select a specific object in the results pane, and choose Actions > Properties.

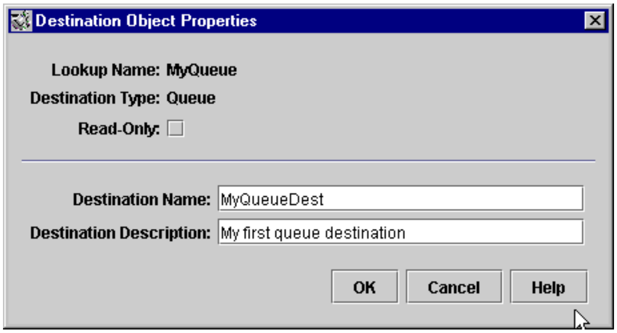
➤ To view or update the properties of a destination object

1. Select Destinations in the navigation pane.

2. Select MyQueue in the results pane.

3. Choose Actions > Properties to view the Destination Object Properties dialog.

   Note that the only value you can change is the destination name and the description. To change the lookup name, you would have to delete the object and then add a new queue administered object with the desired lookup name.



Lookup Name: MyQueue
Destination Type: Queue
Read-Only: □

Destination Name: MyQueueDest
Destination Description: My first queue destination

Updating Console Information

Whether you work with object stores or brokers, you can update the visual display of any element or groups of elements by choosing View > Refresh.

Running the Sample Application

The sample application `SimpleJNDIClient` is provided for use with this tutorial. It uses the administered objects and destination that you created in the foregoing tutorial: a queue destination named `MyQueueDest,` a queue connection factory administered object named `MyQueueConnectionFactory` and a queue administered object named `MyQueue`.

The code creates a simple queue sender and receiver, and sends and receives a "Hello World" message.

➤ To run the `SimpleJNDIClient` application

1. Make sure the `JAVA_HOME` environment variable points to the directory where the JDK is installed.

2. Make the directory that includes the `SimpleJNDIClient` application your current directory; for example:

   `cd IMQ_HOME/demo/jms` (`/usr/demo/imq/jms` on Solaris)

   You should find the `SimpleJNDIClient.class` file present. (If you make changes to the application, you will need to re-compile it using the instructions for compiling a client application in the Quick Start Tutorial of the MQ *Developer's Guide*.)

3. Set the `CLASSPATH` variable to include the current directory containing `SimpleJNDIClient.java` as well as the following jar files: `jms.jar`, `imq.jar`, and `fscontext.jar`.

4. Before you run the application, open the source file, `SimpleJNDIClient.java`, and read through the source.

   It is short, but it is amply documented and it should be fairly clear how it uses the administered objects and destinations you have created using the tutorial.

5. Run the `SimpleJNDIClient` application.

`C:> java SimpleJNDIClient` (Windows)

`% java SimpleJNDIClient file:///tmp` (on Solaris and Linux)

If the application runs successfully, you should see the following output:

```
==================================================
$JAVA SimpleJDNIClient file:///tmp
Usage: java SimpleJDNIClinet [Context.PROVIDER_URL]

On Unix:
        java SimpleJDNIClient file:///tmp
On Windows:
        java SimpleJDNIClient file:///C:Temp

Using Windows default file:///C/Temp for Context.PROVIDER_URL

Publishing a message to Queue: MyQueueDest
Received the following message: Hello World.
==================================================
```

Chapter 5

Starting and Configuring a Broker

After installing Sun™ ONE Message Queue (MQ), you use the `imqbrokerd` command to start a broker. The configuration of the broker instance is governed by a set of configuration files and by options passed with the `imqbrokerd` command, which override corresponding properties in the configuration files.

This chapter explains the syntax of the `imqbrokerd` command and how you use command line options and configuration files to configure the broker instance. In addition, it also describes how you do the following:

- edit a broker instance configuration file
- work with broker clusters
- control logging for the broker

For a description of how to start and use the broker as a Windows service, see "Using a Broker as a Windows Service" on page 235.

Configuration Files

Installed configuration files, which are used to configure the broker, are located in the following directory.

`IMQ_HOME/lib/props/broker`
(`/usr/share/lib/imq/props/broker` on Solaris)

This directory stores the following files:

- A default configuration file that is loaded on startup. This file is called `default.properties` and is not editable. You might need to read this file to determine default settings and to find the exact names of properties you want to change.
- An installation configuration file that contains any properties specified when MQ is installed. This file is called `install.properties`; it cannot be edited after installation.

In addition, the first time you run a broker, an instance configuration file is created that you can use to specify configuration properties for that instance of the broker. This file is maintained by the broker instance in response to administrative commands and can also be edited directly if you're careful. The instance configuration file is stored in the following location:

`IMQ_VARHOME/instances/`*brokerName*`/props/config.properties`
(`/var/imq/instances/`*brokerName*`/props/config.properties` on Solaris)

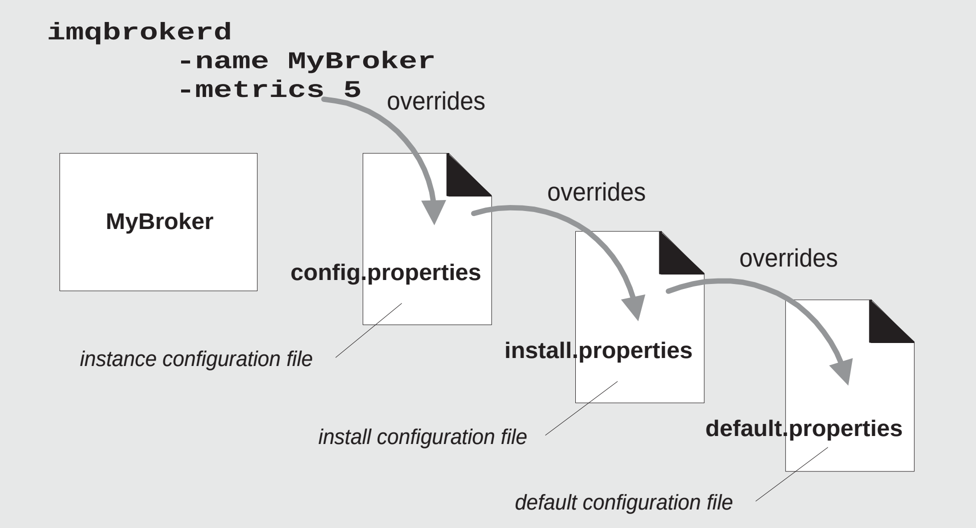
Where *brokerName* is the name of the broker instance (`imqbroker` by default). You can edit an instance configuration file to make configuration changes (see "Editing the Instance Configuration File" on page 116).

If you connect broker instances in a cluster (see "Multi-Broker Clusters (Enterprise Edition)" on page 68) you may also need to use a *cluster configuration file* to specify cluster configuration information. For more information, see "Cluster Configuration Properties" on page 126.

Merging Property Values

At startup, the system merges property values in the different configuration files. It uses values set in the installation and instance configuration files to override values specified in the default configuration file. You can override the resulting values by using `imqbrokerd` command options. This scheme is illustrated in Figure 5-1 on page 115.

Figure 5-1 Broker Configuration Files



Property Naming Syntax

Any MQ property definition in a configuration file uses the following naming syntax:

`propertyName=`*value*`[[,`*value1*`]...]`

For example, the following entry defines the queue type for an auto-create queue:

```
imq.queue.default=single
```

The following entry defines the message expiration timeout value:

```
imq.message.expiration.timeout=90
```

Table 5-1 on page 116 lists the broker configuration properties (and their default values) in alphabetical order.

Editing the Instance Configuration File

The first time a broker instance is run, a `config.properties` file is automatically created. You can edit this instance configuration file to customize the behavior and resource use of the corresponding broker instance.

The broker instance reads the `config.properties` file only at startup. To make permanent changes to the `config.properties` file, you can either

- use administration tools. For information about properties you can set using `imqcmd`, see Table 6-5 on page 147.
- edit the `config.properties` file while the broker instance is shut down; then restart the instance. (On Solaris and Linux platforms, only the user that first started the broker instance has permission to edit the `config.properties` file.)

Table 5-1 lists the broker instance configuration properties (and their default values) in alphabetical order. For more information about the meaning and use of each property, please consult the specified cross-referenced section.

Table 5-1 Broker Instance Configuration Properties

Property Name	Type	Default Value	Reference
`imq.accesscontrol.enabled`	boolean	`true`	Table 2-6 on page 60
`imq.accesscontrol.file.filename`	string	`accesscontrol.properties`	Table 2-6 on page 60
`imq.authentication.basic.user_repository`	string	`file`	Table 2-6 on page 60
`imq.authentication.client.response.timeout`	integer (seconds)	`180`	Table 2-6 on page 60
`imq.authentication.type`	string	`digest`	Table 2-6 on page 60
`imq.autocreate.queue`	boolean	`true`	Table 2-9 on page 67
`imq.autocreate.topic`	boolean	`true`	Table 2-9 on page 67
`imq.cluster.url`	string	`null`	Table 2-10 on page 72
`imq.keystore.`*property_name*			Table 8-8 on page 200
`imq.log.console.output`	string	`ERROR|WARNING`	Table 2-8 on page 63
`imq.log.console.stream`	string	`ERR`	Table 2-8 on page 63

[1] Values that are typed as a *byte string*, can be expressed in bytes, Kbytes, and Mbytes: For example: 1000 means 1000 bytes; 7500b means 7500 bytes; 77k means 77 kilobytes (77 x 1024 = 78848 bytes); 17m means 17 megabytes (17 x 1024 x 1024 = 17825792 bytes)

Table 5-1 Broker Instance Configuration Properties *(Continued)*

Property Name	Type	Default Value	Reference
`imq.log.file.dirpath`	string	`IMQ_VARHOME/instances/`*brokerName*`/log` (`/var/imq/`...on Solaris)	Table 2-8 on page 63
`imq.log.file.name`	string	`log.txt`	Table 2-8 on page 63
`imq.log.file.output`	string	`ALL`	Table 2-8 on page 63
`imq.log.file.rolloverbytes`	integer (bytes)	`0`	Table 2-8 on page 63
`imq.log.file.rolloversecs`	integer (seconds)	`604800`	Table 2-8 on page 63
`imq.log.level`	string	`INFO`	Table 2-8 on page 63
`imq.log.syslog.facility`	string	`LOG_DAEMON`	Table 2-8 on page 63
`imq.log.syslog.logpid`	boolean	`true`	Table 2-8 on page 63
`imq.log.syslog.logconsole`	boolean	`false`	Table 2-8 on page 63
`imq.log.syslog.identity`	string	`imqbrokerd_${imq.`*brokerName*`}`	Table 2-8 on page 63
`imq.log.syslog.output`	string	`ERROR`	Table 2-8 on page 63
`imq.message.expiration.interval`	integer (seconds)	`60`	Table 2-4 on page 53
`imq.message.max_size`	byte string [1] 0 (no limit)	`70m`	Table 2-4 on page 53
`imq.metrics.enabled`	boolean	`true`	Table 2-8 on page 63
`imq.metrics.interval`	integer (seconds)	`0`	Table 2-8 on page 63
`imq.passfile.enabled`	boolean	`false`	Table 2-6 on page 60
`imq.passfile.dirpath`	string	`IMQ_HOME/etc` (`/etc/imq` on Solaris)	Table 2-6 on page 60
`imq.passfile.name`	string	`passfile`	Table 2-6 on page 60
`imq.persist.file.destination.file.size`	byte string [1]	`1m`	Table 2-5 on page 56
`imq.persist.file.message.cleanup`	boolean	`false`	Table 2-5 on page 56

[1] Values that are typed as a *byte string*, can be expressed in bytes, Kbytes, and Mbytes: For example: 1000 means 1000 bytes; 7500b means 7500 bytes; 77k means 77 kilobytes (77 x 1024 = 78848 bytes); 17m means 17 megabytes (17 x 1024 x 1024 = 17825792 bytes)

Table 5-1 Broker Instance Configuration Properties *(Continued)*

Property Name	Type	Default Value	Reference
`imq.persist.file.message.fdpool.limit`	integer	`25` (Solaris & Linux) `1024` (Windows)	Table 2-5 on page 56
`imq.persist.file.message.filepool.cleanratio`	integer	`0`	Table 2-5 on page 56
`imq.persist.file.message.filepool.limit`	integer	`10000`	Table 2-5 on page 56
`imq.persist.file.sync.enabled`	boolean	`false`	Table 2-5 on page 56
`imq.persist.jdbc.`*property_name*			Table A-1 on page 207
`imq.persist.store`	string	`file`	Table 2-5 on page 56
`imq.portmapper.port`	integer	`7676`	Table 2-3 on page 49
`imq.queue.deliverypolicy`	string	`single`	Table 2-9 on page 67
`imq.redelivered.optimization`	boolean	`true`	Table 2-4 on page 53

`imq.`*resource_state*`.threshold`	integer (percent)	`0 (green)`
		`60 (yellow)`
		`75(orange)`
		`90 (red)`
			Table 2-4 on page 53

`imq.service.activelist`	list	`jms,admin`	Table 2-3 on page 49
`imq.`*service_name*`.accesscontrol.enabled`	boolean	inherits value from system-wide property	Table 2-6 on page 60
`imq.`*service_name*`.accesscontrol.file.filename`	string	inherits value from system-wide property	Table 2-6 on page 60
`imq.`*service_name*`.authentication.type`	string	inherits value from system-wide property	Table 2-6 on page 60
`imq.`*service_name*`.max_threads`	integer	`1000` (jms) `500` (ssljms) `500` (httpjms) `500` (httpsjms) `50` (admin)	Table 2-3 on page 49

[1] Values that are typed as a *byte string*, can be expressed in bytes, Kbytes, and Mbytes: For example: 1000 means 1000 bytes; 7500b means 7500 bytes; 77k means 77 kilobytes (77 x 1024 = 78848 bytes); 17m means 17 megabytes (17 x 1024 x 1024 = 17825792 bytes)

Table 5-1 Broker Instance Configuration Properties *(Continued)*

Property Name	Type	Default Value	Reference
`imq.`*service_name*`.min_threads`	integer	`10` (jms) `10` (ssljms) `10` (httpjms) `10` (httpsjms) `4` (admin)	Table 2-3 on page 49
`imq.`*service_name*`.`*protocol type*`.hostname`	string	`null`	Table 2-3 on page 49
`imq.`*service_name*`.`*protocol type*`.port`	integer	`0`	Table 2-3 on page 49
`imq.`*service_name*`.threadpool_model`	string	`dedicated` (jms) `dedicated` (ssljms) `dedicated` (httpjms) `dedicated` (httpsjms) `dedicated` (admin)	Table 2-3 on page 49
`imq.shared.connectionMonitor_limit`	integer	`512` (Solaris & Linux) 64 (Windows)	Table 2-3 on page 49
`imq.system.max_count`	integer, 0 (no limit)	`0`	Table 2-4 on page 53
`imq.system.max_size`	byte string[1], 0 (no limit)	`0`	Table 2-4 on page 53
`imq.transaction.autorollback`	boolean	`false`	Table 2-4 on page 53
`imq.user_repository.ldap.`*property_name*			Table 8-5 on page 189

[1] Values that are typed as a *byte string*, can be expressed in bytes, Kbytes, and Mbytes: For example: 1000 means 1000 bytes; 7500b means 7500 bytes; 77k means 77 kilobytes (77 x 1024 = 78848 bytes); 17m means 17 megabytes (17 x 1024 x 1024 = 17825792 bytes)

Starting a Broker

To start a broker instance use the `imqbrokerd` command.

NOTE You cannot start a broker instance using the Administration Console (imqadmin) or the Command Utility (imqcmd). The broker instance must already be running to use these MQ administration tools.

To override one or more property values, specify a valid `imqbrokerd` command-line option. Command-line options override values in the broker configuration files, but only for the current broker session: command line options are not written to the instance configuration file.

The syntax of the `imqbrokerd` command is as follows (options and arguments are separated by a space):

```
imqbrokerd [[ -Dproperty=value]...]
    [ -backup fileName]
    [ -cluster “[broker] [[,broker]...]”
    [ -dbuser userName] [ -dbpassword password]
    [ -force]
    [ -h]
    [ -javahome path | -jrehome path]
    [ -ldappassword password]
    [ -license name]
    [ -loglevel level]
    [ -metrics number]
    [ -name brokerName ] [ -port number]
    [ -shared]
    [ -password keypassword] [ -passfile fileName]
    [ -remove instance]
    [ -reset data]
    [ -restore fileName]
    [ -shared]
    [ -silent] [ -tty]
    [ -version] [ -vmargs arg [[arg]...]
```

For example, to start a broker that uses the default broker name and configuration, use the following command:

```
imqbrokerd
```

This starts a default instance of a broker (named `imqbroker`) on the local machine with the Port Mapper at port 7676.

NOTE On Solaris, you can configure the broker to automatically restart after an abnormally exit, by setting the `RESTART` property in the `/etc/imq/imqborkerd.conf` configuration file to `YES`.

NOTE On Solaris and Linux platforms, permissions on the directories containing configuration information and persistent data depend on the umask of the user that starts the broker instance the first time. Hence, for the broker instance to function properly, it must be started subsequently only by the original user.

Table 5-2 describes the options to the `imqbrokerd` command and describes the configuration properties, if any, affected by each option.

Table 5-2 `imqbrokerd` Options

Option	Properties Affected	Description
`-backup` *fileName*	None affected.	Applies only to broker clusters. Backs up a Master Broker's configuration change record to the specified file. See "Backing up the Master Broker's Configuration Change Record" on page 130.
`-cluster"[`*broker*`][[,`*broker*`]...]"` *broker* is either • *host*[:*port*] • [*host*]:*port*	Sets `imq.cluster.brokerlist` to the list of brokers to which to connect.	Applies only to broker clusters. Connects to all the brokers on the specified hosts and ports. This list is merged with the list in the `imq.cluster.brokerlist` property. If you don't specify a value for *host*, `localhost` is used. If you don't specify a value for *port*, the value 7676 is used. See "Working With Clusters (Enterprise Edition)" on page 126 for more information on how to use this option to connect multiple brokers.
`-dbpassword` *password*	Sets `imq.persist.jdbc.password` to specified password	Specifies the password for a plugged-in JDBC-compliant data store. See Appendix A, "Setting Up Plugged-in Persistence."
`-dbuser` *userName*	Sets `imq.persist.jdbc.user` to specified user name	Specifies the user name for a plugged-in JDBC-compliant database. See Appendix A, "Setting Up Plugged-in Persistence."

Table 5-2 `imqbrokerd` Options *(Continued)*

Option	Properties Affected	Description
`-D`*property*`=`*value*	Sets system properties. Overrides corresponding property value in instance configuration file.	Sets the specified property to the specified value. See Table 5-1 on page 116 for broker configuration properties. **Caution:** Be careful to check the spelling and formatting of properties set with the D option. If you pass incorrect values, the system will not warn you, and MQ will not be able to set them.
`-force`	None affected.	Performs action without user confirmation. This option applies only to the `-remove instance` option, which normally requires confirmation.
`-h`	None affected.	Displays help. Nothing else on the command line is executed.
`-javahome` *path*	None affected.	Specifies the path to an alternate Java 2- compatible JDK. The default is to use the bundled runtime.
`-jrehome` *path*	None affected.	Specifies the path to a Java 2 JRE.
`-ldappassword` *password*	Sets `imq.user_repository.ldap.password` to specified password	Specifies the password for accessing a LDAP user repository. See "Using an LDAP Server for a User Repository" on page 189.
`-license [`*name*`]`	None affected.	Specifies the license to load, if different from the default for your MQ product edition. If you don't specify a license name, this lists all licenses installed on the system. Depending on the installed MQ edition, the values for *name* are `pe` (Platform Edition—basic features), `try` (Platform Edition—90-day trial enterprise features), and `unl` (Enterprise Edition). See "Product Editions" on page 26.
`-loglevel` *level*	Sets `imq.broker.log.level` to the specified level.	Specifies the logging level as being one of `NONE`, `ERROR`, `WARNING`, or `INFO`. The default value is `INFO`. For more information, see "Logger" on page 61.

Table 5-2 `imqbrokerd` Options *(Continued)*

Option	Properties Affected	Description
`-metrics` *int*	Sets `imq.metrics.report.interval` to the specified number of seconds.	Specifies that metrics be reported at an interval specified in seconds.
`-name` *brokerName*	Sets `imq.instancename` to the specified name.	Specifies the instance name of this broker and uses the corresponding instance configuration file. If you do not specify a broker name, the name of the file is set to `imqbroker`. **Note**: If you run more than one instance of a broker on the same host, each must have a unique name.
`-passfile` *fileName*	Sets `imq.passfile.enabled` to `true`. Sets jmq.`passfile.dirpath` to the path that contains the file. Sets `imq.passfile.name` to the name of the file.	Specifies the name of the file from which to read the passwords for the SSL keystore, LDAP user repository, or JDBC-compliant database. For more information, see "Using a Passfile" on page 204.
`-password` *keypassword*	Sets `imq.keystore.password` to the specified password.	Specifies the password for the SSL certificate keystore. For more information, see "Security Manager" on page 57.
`-port` *number*	Sets `imq.portmapper.port` to the specified number.	Specifies the broker's Port Mapper port number. By default, this is set to 7676. To run two instances of a broker on the same server, each broker's Port Mapper must have a different port number. JMS clients connect to the broker instance using this port number.
`-remove instance`	None affected.	Causes the broker instance to be removed: deletes the instance configuration file, log files, persistent store, and other files and directories associated with the instance. Requires user confirmation unless `-force` option is also specified.

Table 5-2 `imqbrokerd` Options *(Continued)*

Option	Properties Affected	Description

`-reset store|`
`messages|`
`durables|`
`props`
	None affected.	Resets the data store (or a subset of the data store) or the configuration properties of a broker instance, depending on the argument given.
		Resetting the data store clears out all persistent data, including persistent messages, durable subscriptions, and transaction information. This allows you to start the broker instance with a clean slate. You can also clear only all persistent messages or only all durable subscriptions. (If you do not want the persistent store to be reset on subsequent starts, then re-start the broker instance without using the `-reset` option.) For more information, see "Persistence Manager" on page 54.
		Resetting the broker's properties, replaces the existing instance configuration file (`config.properties`) with an empty file: all properties assume default values.

`-restore` *fileName*	None affected.	Applies only to broker clusters. Replaces the Master Broker's configuration change record with the specified backup file. This file must have been previously created using the `-backup` option. See "Restoring the Master Broker's Configuration Change Record" on page 131.
`-shared`	Sets `imq.jms.threadpool_model` to `shared`.	Specifies that the jms connection service be implemented using the shared threadpool model, in which threads are shared among connections to increase the number of connections supported by a broker instance. For more information, see "Connection Services" on page 46.
`-silent`	Sets `imq.log.console.output` to `NONE`.	Turns off logging to the console.
`-tty`	Sets `imq.log.console.output to ALL`	Specifies that all messages be displayed to the console. By default only `WARNING` and `ERROR` level messages are displayed.

Table 5-2 `imqbrokerd` Options *(Continued)*

Option	Properties Affected	Description
`-version`	None affected.	Displays the version number of the installed product.
`-vmargs` *arg* `[[`*arg*`]...]`	None affected	Specifies arguments to pass to the Java VM. Separate arguments with spaces. If you want to pass more than one argument or if an argument contains a space, use enclosing quotation marks. For example: `imqbrokerd -tty -vmargs "-Xmx128m -Xincgc"`

Working With Clusters (Enterprise Edition)

This section describes the properties you use to configure multi-broker clusters, describes a couple of methods of connecting brokers, and explains how you manage clusters. For an introduction to clusters, see "Multi-Broker Clusters (Enterprise Edition)" on page 68.

When working with clusters, make sure that you synchronize clocks among the hosts of all brokers in a cluster.

Cluster Configuration Properties

When you connect brokers into a cluster, all the connected brokers must specify the same values for cluster configuration properties. These properties describe the participation of the brokers in a cluster. Table 5-3 summarizes the cluster-related configuration properties.

Table 5-3 Cluster Configuration Properties

Property	Description
`imq.cluster.brokerlist`	Specifies all brokers in a cluster in a comma-separated list; each item specifies the host and port of a broker. For example: `host1:3000, host2:8000, ctrhost`
`imq.cluster.masterbroker`	Specifies the host and port of the Master Broker. Set this value for production environments. For example, `ctrhost:7676`

`imq.cluster.url`	Specifies the location of the cluster configuration file.
	For example:
	`http://webserver/imq/cluster.properties`
	`file:/net/mfsserver/imq/cluster.properies`

`imq.cluster.port`	For *each* broker within a cluster, can be used to specify the port number for the cluster connection service. The cluster connection service is used for internal communication between brokers in a cluster. Default: `0` (port is dynamically allocated)

Table 5-3 Cluster Configuration Properties *(Continued)*

Property	Description
`imq.cluster.hostname`	For *each* broker within a cluster, can be used to specify the host (hostname or IP address) to which the cluster connection service binds if there is more than one host available (for example, if there is more than one network interface card in a computer). The cluster connection service is used for internal communication between brokers in a cluster. Default: `null` (all available hosts)

You can use one of two methods to set cluster properties:

- You set the cluster-related configuration properties in each broker's instance configuration file (or in the command line that starts each broker). For example, to connect broker A (on `host1`, port 7676), broker B (on `host2`, port 5000) and broker C (on `ctrlhost`, port 7676), the instance configuration file for brokers A, B, and C would need to set the following property.

  ```
  imq.cluster.brokerlist=host1, host2:5000, ctrlhost
  ```

  If you decide to change a cluster configuration, this method requires you to update cluster-related properties in all the brokers

- You set cluster configuration properties in one central cluster configuration file. These properties might include the list of brokers to be connected (`imq.cluster.brokerlist`) and optionally, the address of the Master Broker (`imq.cluster.masterbroker`).

  If you use this method, you must also set the `imq.cluster.url` property (for every broker in the cluster) to point to the location of the cluster configuration file. From the point of view of easy maintenance, this is the recommended method of cluster configuration.

The following code sample shows the contents of a cluster configuration file. Both `host1` and `ctrlhost` are running on the default port. These properties specify that `host1` and `ctrlhost` are connected in a cluster and that `ctrlhost` is the Master Broker.

```
imq.cluster.brokerlist=host1,host2:5000,ctrlhost
imq.cluster.masterbroker=ctrlhost
```

The instance configuration file for each broker connected in this cluster, must then contain the url of the cluster configuration file; for example:

```
imq.cluster.url=file:/home/cluster.properties
```

Connecting Brokers

This section describes two methods of connecting brokers into a clusters. No matter which method you use, each broker that you start attempts to connect to the other brokers every five seconds; that attempt will succeed once the other brokers in the cluster are started up.

If you connect brokers into a cluster, it is not necessary to start the Master Broker first. If a broker in the cluster starts before the Master Broker, it will remain in a suspended state, rejecting client connections. When the Master Broker starts, the suspended broker will automatically become fully functional.

Method 1: No Cluster Configuration File

➤ **To connect brokers into a cluster**

1. Use the `-cluster` option to the `imqbrokerd` command that starts a broker, and specify the complete list of brokers (to connect to) as an argument to the `-cluster` option.

2. Do this for each broker you want to connect to the cluster when you start that broker.

   For example, the following command starts a new broker and connects it to the broker running on the default port on host1, the broker running on port 7677 on host2 and the broker running on port 7678 on localhost.

   ```
   imqbrokerd -cluster host1,host2:7677,:7678
   ```

Method 2: Using a Cluster Configuration File

It is also possible to create a cluster configuration file that specifies the list of brokers to be connected (and optionally, the address of the Master Broker). This method of defining clusters is better suited for production systems. Remember, that each broker in the cluster must set the value of the `imq.cluster.url` property to point to the cluster configuration file.

Adding Brokers to Clusters

Once you have set up a broker cluster, you might need to add a new broker or restart a broker that is already part of the cluster.

To add a new broker to an existing cluster, you can do one of the following:

If you are not using a cluster configuration file, when you start the new broker, specify the `imq.cluster.brokerlist` and (if necessary) the `imq.cluster.masterbroker` properties on the command line using the `-D` option.

➤ **To add a broker to a cluster if you are using a cluster configuration file**

1. Add the new broker to the `imq.cluster.brokerlist` property in the cluster configuration file.
2. Issue the following command to any broker in the cluster.

   ```
   imqcmd reload cls
   ```

   This forces all the brokers to reload the `imq.cluster.brokerlist` property and to make sure that all persistent information for brokers in the cluster is up to date.

Restarting a Broker in a Cluster

To restart a broker that is already a member of a cluster, you can do one of the following:

- If the cluster is defined using a cluster configuration file, use the -D option to specify the `imq.cluster.url` property on the command line used to start the broker.
- If the cluster is not defined using a cluster configuration file, when you start the new broker, specify the `imq.cluster.brokerlist` (and if necessary the `imq.cluster.masterbroker`) properties on the command line using the -D option. If the cluster does not include a Master Broker, you can simply use the `-cluster` option to specify the list of brokers in the cluster when you start the new broker.

Removing a Broker from a Cluster

Take note of the following when removing a broker from a cluster:

- If the brokers A, B, and C were all started using the following command line, then just restarting A will not remove it from the cluster.

  ```
  imqbrokerd -cluster A,B,C
  ```

  Instead, you need to restart all the other brokers with the following command line:

  ```
  imqbrokerd -cluster B,C
  ```

  Then, you need to start broker A without specifying the `-cluster` option.

- If the list of brokers was specified using a cluster configuration file, then you will need to do the following:
  - Remove mention of the broker from the configuration file.
  - Change or remove the `imq.cluster.url` property for the broker that is being removed so that it no longer uses the common properties.
  - Use the `imqcmd reload cls` command to force all the brokers to reload their cluster configuration and thereby reconfigure the cluster.

Backing up the Master Broker's Configuration Change Record

Each cluster can have one Master Broker that keeps track of any changes in the persistent state of the cluster: this includes durable subscriptions and physical destinations created by the administrator. All brokers consult the Master Broker during startup in order to synchronize information about these persistent objects. Consequently, the failure of the Master Broker can cripple the entire cluster. For this reason, it is important to backup the Master Broker's change record periodically by using the `-backup` option of the `imqbrokerd` command. For example,

```
imqbrokerd -backup mybackuplog
```

It is important you do this in a timely manner. Restoring a very old backup can result in loss of information: any persistent objects created since the backup was last done will be lost.

Restoring the Master Broker's Configuration Change Record

➤ **To restore the Master Broker in case of failure**

1. Shut down all the brokers in the cluster.
2. Restore the Master Broker's configuration change record using the following command:

   ```
   imqbrokerd  -restore mybackuplog
   ```

3. If you assign a new name or port number to the Master Broker, you must update the cluster configuration file to specify that the Master Broker is part of the cluster and to specify its new name (using the property `imq.cluster.masterbroker`).
4. Restart all the brokers.

The restoration of the broker will inevitably result in some stale data being reloaded into the broker's configuration change record; however, doing frequent periodic backups, as described in the previous section, should minimize this problem.

Because the Master Broker keeps track of the entire history of changes to persistent objects, its database can grow significantly over a period of time. The backup and restore operations have the positive effect of compressing and optimizing this database.

Logging

This section describes the default logging configuration for the broker and explains how you can change that configuration in order to redirect log information to alternate output channels, to change rollover criteria, and to report broker metrics. For an introduction to logging, see "Logger" on page 61.

Default Logging Configuration

When you start the broker, it is automatically configured to save log output to a set of rolling log files located at

`IMQ_VARHOME/instances/`*brokerName*`/log/`
(`/var/imq/instances/`*brokerName*`/log/` on Solaris)

The log files are simple text files. They are named as follows, from earliest to latest:

```
log.txt
log_1.txt
log_2.txt
...
log_9.txt
```

By default, log files are rolled over once a week; the system maintains nine backup files.

- To change the directory in which the log files are kept, set the property `imq.log.file.dirpath` to the desired path.
- To change the root name of the log files from `log` to something else, set the `imq.log.file.filename` property.

The broker supports three log categories: `ERROR`, `WARNING`, `INFO` (see Table 2-7 on page 62). Setting a logging level gathers messages for all levels up to and including that level. The default log level is `INFO`. This means that `ERROR`, `WARNING`, and `INFO` messages are logged.

Log Message Format

Logged messages consist of a timestamp, message code, and the message itself. The volume of information varies with the log level you have set. The following is an example of an `INFO` message.

```
[13/Sep/2000:16:13:36 PDT] B1004 Starting the broker service
using tcp [ 25374,100] with min threads 50 and max threads of 500
```

Changing the Logger Configuration

All Logger properties are described in Table 2-8 on page 63.

➤ **To change the Logger configuration for a broker**

1. Set the log level.
2. Set the output channel (file, console, or both) for one or more logging categories.
3. If you log output to a file, configure the rollover criteria for the file.

You complete these steps by setting Logger properties. You can do this in one of two ways:

- Change or add Logger properties in the `config.properties` file for a broker before you start the broker.
- Specify Logger command line options in the `imqbrokerd` command that starts the broker. You can also use the broker option `-D` to change Logger properties (or *any* broker property).

Options passed on the command line override properties specified in the broker instance configuration files. Table 5-4 lists the `imqbrokerd` options that affect logging.

Table 5-4 `imqbrokerd` Logger Options and Corresponding Properties

imqbrokerd Options	**Description**
`-metrics` *number*	Specifies the interval (in seconds) at which metrics information is gathered.
`-loglevel` *level*	Sets the log level to one of `ERROR`, `WARNING`, `INFO`.
`-silent`	Turns off logging to the console
`-tty`	Sends all messages to the console. By default only `WARNING` and `ERROR` level messages are displayed.

The following sections describe how you can change the default configuration in order to do the following:

- change the output channel (the destination of log messages)
- change rollover criteria
- log broker metrics information

Changing the Output Channel

By default, error and warning messages are displayed on the terminal as well as being logged to a log file. (On Solaris error messages are also written to the system's syslog daemon.)

You can change the output channel for log messages in the following ways:

- To have *all* log categories (for a given level) output displayed on the screen, use the `-tty` option to the `imqbrokerd` command.
- To prevent log output from being displayed on the screen, use the `-silent` option to the `imqbrokerd` command.
- Use the `imq.log.file.output` property to specify which categories of logging information should be written to the log file. For example,

  ```
  imq.log.file.output=ERROR
  ```
- Use the `imq.log.console.output` property to specify which categories of logging information should be written to the console. For example,

  ```
  imq.log.console.output=INFO
  ```
- On Solaris, use the `imq.log.syslog.output` property to specify which categories of logging information should be written to Solaris syslog. For example,

  ```
  imq.log.syslog.output=NONE
  ```

NOTE Before changing the destination of log messages, you must make sure that logging is set at the level that corresponds to the log category you are mapping to the output channel. For example, if you set the log level to `ERROR` and then set the `imq.log.console.output` property to `WARNING`, no messages will be logged because you have not enabled the logging of those level messages.

Changing Rollover Criteria

There are two criteria for rolling over log files: time and size. The default is to use a time criteria and roll over files every seven days.

- To change the time interval, you need to change the property `imq.log.file.rolloversecs`. For example, the following property definition changes the time interval to ten days:

  ```
  imq.log.file.rolloversecs=864000
  ```

- To change the rollover criteria to depend on file size, you need to set the `imq.log.file.rolloverbytes` property. For example, the following definition directs the broker to rollover files after they reach a limit of 500,000 bytes

  ```
  imq.log.file.rolloverbytes=500000
  ```

If you set both the time-related and the size-related rollover properties, the first limit reached will trigger the rollover. As noted before, the broker maintains up to nine rollover files.

Logging Broker Performance Metrics

The broker's default configuration, includes the following settings:

- `imq.metrics.enabled=true`
- `imq.metrics.interval=0`
- `imq.log.level=INFO`

As a result of these settings, the broker gathers performance metrics for the broker as well as for active connection services, but it does not generate metrics reports.

You can have the broker generate metrics reports in one of two ways:

- Use the `-metrics` option to the `imqbrokerd` command and specify the interval (in seconds) at which the broker generates reports.
- Set the `imq.metrics.interval` property to the interval (in seconds) at which you want the broker to generate reports.

Because metrics reports are included in the `INFO` category, metric reports, by default, are written to the log file output channel.

The following shows sample metrics information:

```
[31/Jan/2001:15:00:50 PST]
Connections: 0 JVM Heap: 6291456 bytes (5186320 free)
      In: 0 mesgs (0bytes) 0 pkts (0 bytes)
     Out: 0 mesgs (0bytes) 0 pkts (0 bytes)
 Rate In: 0 msgs/sec (0 bytes/sec) 0 pkts/sec (0 bytes/sec)
Rate Out: 0 msgs/sec (0 bytes/sec) 0 pkts/sec (0 bytes/sec)
```

Table 5-5 describes the meaning of the metrics generated for each connection service.

Table 5-5 Metrics Gathered for Connection Services

Metrics	Description
Pkts in (total)	Total number of packets read by the broker since the last reset. This includes MQ protocol packets, not just JMS messages.
Pkts out (total)	Total number of packets written by the broker since the last reset.
JMS Messages in (total)	Total number of JMS messages read by the broker since last reset.
JMS Messages out (total)	Total number of JMS messages written by the Broker since last reset.
Message Bytes in (total)	Total number of message bytes read by the Broker since last reset.
Message Bytes out (total)	Total number of message bytes written.
Current # connections	Current number of open connections.

Table 5-6 describes the metrics gathered and reported for each broker.

Table 5-6 Metrics Gathered for Each Broker

Metrics	Description
VM heap size (bytes)	Maximum size of the Java VM heap.
VM heap free space (bytes)	Amount of free space left in the Java VM heap.

NOTE This information is also available via the `imqcmd metrics` command.

Chapter 6

Broker and Application Management

This chapter explains how to perform tasks related to managing the broker and the services it provides. Some of these tasks are independent of any particular client application. These include:

- controlling the broker's state: you can pause, resume, shutdown, and restart the broker.
- querying and updating broker properties
- querying and updating connection services
- allocating and managing resources
- managing connection services

Other broker tasks are performed on behalf of specific applications; these include managing physical destinations, durable subscriptions, and transactions:

- MQ messages are routed to their receivers or subscribers by way of broker destinations. You are responsible for creating these destinations on the broker.
- MQ allocates and maintains resources for durable subscribers even when clients that have durable subscriptions become inactive. You use the MQ Command tool to get information about durable subscriptions and to destroy durable subscriptions or purge their messages in order to save MQ resources.
- MQ transactions and distributed transactions are tracked by a broker. You might need to manually commit or roll back transactions if a failure takes place.

This chapter explains how you use the Command utility (`imqcmd`) to perform all these tasks. You can accomplish many of these same tasks by using the Administration Console, the graphical interface to the MQ message server. For more information, see Chapter 4, "Administration Console Tutorial."

Command Utility

The Command utility allows you to manage the broker and the services it provides. This section describes the basic `imqcmd` command syntax, provides a listing of subcommands, and summarizes `imqcmd` options. Subsequent sections explain how you use these commands to accomplish specific tasks.

Syntax of Command

The general syntax of the `imqcmd` command is as follows:

```
imqcmd subcommand argument [options]
imqcmd -h|H
imqcmd -v
```

Note that if you specify the -v, -h, or -H options, no subcommands specified on the command line are executed. For example, if you enter the following command, version information is displayed but the `restart` subcommand is not executed.

```
imqcmd restart bkr -v
```

imqcmd Subcommands

The Command utility (`imqcmd`) includes the subcommands listed in Table 6-1:

Table 6-1 `imqcmd` Subcommands

Subcommand and Argument	Description
`create dst`	Creates a destination.
`commit txn`	Commits a transaction.
`destroy dst`	Destroys a destination.
`destroy dur`	Destroys a durable subscription.
`list dst`	Lists destinations on the broker.
`list dur`	Lists durable subscriptions on the topic.
`list svc`	Lists services on the broker.
`list txn`	Lists transactions on the broker.
`metrics bkr`	Displays broker metrics.
`metrics svc`	Displays service metrics.

Table 6-1 `imqcmd` Subcommands *(Continued)*

Subcommand and Argument	Description
`pause bkr`	Pauses all services on the broker.
`pause svc`	Pauses one service.
`purge dst`	Purges all messages on a destination without destroying the destination.
`purge dur`	urges all messages on a durable subscription without destroying the durable subscription.
`query bkr`	Queries and display information on a broker.
`query dst`	Queries and display information on a destination.
`query svc`	Queries and display information on a service.
`query txn`	Queries and display information on a transaction.
`reload cls`	Reloads broker cluster configuration.
`restart bkr`	Restarts the current running broker instance. Cannot be used to start a new broker instance.
`resume bkr`	Resumes all services on the broker.
`resume svc`	Resumes one service.
`rollback txn`	Rolls back a transaction.
`shutdown bkr`	Shuts down the broker instance. Can be subsequently started using the `imqbrokerd` command, but not the `restart bkr` subcommand of `imqcmd`.
`update bkr`	Updates attributes of a broker.
`update dst`	Updates attributes of a destination.
`update svc`	Updates attributes of a service.

Summary of imqcmd Options

Table 6-2 lists the options to the `imqcmd` command. For a discussion of their use, see the following task-based sections.

Table 6-2 `imqcmd` Options

Option	Description
`-b` *hostName:port*	Specifies the name of the broker's host and its port number. The default value is `localhost:7676`. To specify port only: `-b :7878` To specify name only: `-b somehost`
`-c` *clientID*	Specifies the ID of the durable subscriber to a topic. For more information, see "Managing Durable Subscriptions" on page 155.
`-d` *topicName*	Specifies the name of the topic. Used with the `list dur` and `destroy dur` subcommands. See "Managing Durable Subscriptions" on page 155.
`-f`	Performs action without user confirmation.
`-h`	Displays usage help. Nothing else on the command line is executed.
`-H`	Displays usage help, attribute list, and examples. Nothing else on the command line is executed.
`-int` *interval*	Specifies the interval, in seconds, at which `imqcmd` displays broker metrics. (Used with the `metrics` subcommand.)
`-javahome`	Specifies an alternate Java 2 compatible runtime to use (default is to use the runtime bundled with the product).
`-m` *metricType*	Specifies the type of metric information to display. Type can be one of the following `ttl` Total of messages in and out of the broker (default). `rts` Provides the same information as ttl, but specifies the number of messages per second. `cxn` Connections, virtual memory heap, threads Use this option with the `metrics bkr` or `metrics svc` subcommand. The following command displays cxn-type metrics for the default broker every five seconds. `imqcmd metrics bkr -m cxn -int 5`

Table 6-2 `imqcmd` Options *(Continued)*

Option	Description
`-n` *argumentName*	Specifies the name of the subcommand argument. Depending on the subcommand, this might be the name of a service, a physical destination, a durable subscription, or a transaction ID.
`-o` *attribute=value*	Specifies the value of an attribute. Depending on the subcommand argument, this might be the attribute of a broker (see "Querying and Updating Broker Properties" on page 145), service (see "Managing Connection Services" on page 148), or destination (see "Managing Destinations" on page 152).
`-secure`	Specifies a secure administration connection to the broker using the `ssladmin` connection service (see "Step 4. Configuring and Running SSL-based Clients,""Command Utility (imqcmd)" on page 202).
`-p` *password*	Specifies your (the administrator's) password. If you omit this value, you will be prompted for it.
`-s`	Silent mode. No output will be displayed.
`-t` *destinationType*	Specifies the type of a destination: `t` (topic) or `q` (queue).
`-tmp`	Displays temporary destinations.
`-u` *name*	Specifies your (the administrator's) name. If you omit this value, you will be prompted for it.
`-v`	Displays version information. Nothing else on the command line is executed.

You must specify the options for host name and port number (`-b`), user name (`-u`) password (`-p`), and secure connection (`-secure`) *each time* you issue a `imqcmd` subcommand. If you don't specify the host name and port number, it uses the default values. If you don't specify user name and password information, you will be prompted for them. If you don't specify `-secure`, then the connection will not be secure.

Prerequisites to Using imqcmd

In order to use imqcmd commands to manage the broker, you must do the following:

- Start the broker using the imqbrokerd command.

  See "Starting a Broker" on page 120. You can use the Command utility only to administer brokers that are already running; you cannot use it to start a broker.

- Specify the target broker using the -b option unless the broker is running on the local host, on port 7676.

- Specify the proper administrator user name and password. If you do not do this, you will be prompted for it. Either way, be aware that every operation you perform using imqcmd will be authenticated against a user repository.

  When you install MQ, a default flat-file user repository is installed. The file is named IMQ_HOME/etc/passwd (/etc/imq/passwd on Solaris). The repository is shipped with two entries: one for an admin user and one for a guest user. These entries allow you to connect to the broker without doing any additional work. For example, if you are just testing MQ, you can run the utility using your default user name and password (admin/admin).

  If you are setting up a production system, you will need to do some additional work to authenticate and authorize users. You also have the option of using an existing LDAP directory server for your user repository. For more information, see "Authenticating Users" on page 182.

Examples

The following command lists the properties of the broker running on localhost at port 7676:

```
imqcmd query bkr -u admin -p admin
```

The following command lists the properties of the broker running on myserver at port 1564; the user's name is alladin, the user's password is abracadabra.

```
imqcmd query bkr -b myserver:1564 -u alladin -p abracadabra
```

Assuming that the user name alladin was assigned to the admin group, you will be connected as an admin client to the specified broker.

Controlling the Broker’s State

After you start the broker, you can use the following `imqcmd` subcommands to control the state of the broker.

- Pausing the broker

  Pausing the broker suspends the broker service threads which causes the broker to stop listening on the ports. You can then perform any administration tasks needed to regulate the flow of messages to the broker. For example, if a particular destination is bombarded with messages, you can pause the broker and take any of the following actions that might help you fix the problem: trace the source of the messages, limit the size of the destination, or destroy the destination.

  The following command pauses the broker running on `myhost` at port 1588.

  ```
  imqcmd pause bkr -b myhost:1588
  ```

- Resuming the broker

  Resuming the broker reactivates the broker’s service threads and the broker resumes listening on the ports. The following command resumes the broker running on `localhost` at port 7676.

  ```
  imqcmd resume bkr
  ```

- Shutting down the broker

  Shutting down the broker terminates the broker process. This is a graceful termination: the broker stops accepting new connections and messages, it completes delivery of existing messages, and it terminates the broker process. The following command shuts down the broker running on `ctrlsrv` at port 1572

  ```
  imqcmd shutdown bkr -b ctrlsrv:1572
  ```

- Restarting the broker

  Shuts down and restarts the broker. The following command restarts the broker running on `localhost` at port 7676:

  ```
  imqcmd restart bkr
  ```

Table 6-3 summarizes the `imqcmd` subcommands used to control the broker. Remember that you must specify the broker host name and port number unless you are targeting the broker running on `localhost` at port 7676.

Table 6-3 `imqcmd` Subcommands Used to Control the Broker

Subcommand	Description
`pause bkr [-b` *hostName:port*`]`	Pauses the default broker or a broker at the specified host and port.
`resume bkr [-b` *hostName:port*`]`	Resumes the default broker or a broker at the specified host and port.
`shutdown bkr [-b` *hostName:port*`]`	Shuts down the default broker or a broker at the specified host and port.
`restart bkr [-b` *hostName:port*`]`	Shuts down and restart the default broker or a broker at the specified host and port. Note that this command restarts the broker using the options specified when the broker was first started. If you want different options to be in effect, you must shutdown the broker and then start it again, specifying the options you want.

Querying and Updating Broker Properties

The Command utility includes subcommands that you can use to get information about the broker and to update broker properties. Table 6-4 lists these subcommands.

Table 6-4 `imqcmd` Subcommands Used to Get Information and to Update Broker

Subcommand Syntax	Description
`query bkr -b` *hostName:port*	Lists the current settings of properties of the default broker or a broker at the specified host and port. Also shows the list of running brokers (in a multi-broker cluster) that are connected to the specified broker.
`reload cls`	Applies only to broker clusters. Forces all the brokers in a cluster to reload the `imq.cluster.brokerlist` property and update cluster information. See "Adding Brokers to Clusters" on page 129 for more information.

`update bkr [-b` *hostName:port*`]`
`-o` *attribute=value*
`[-o` *attribute=value1*`]...`
	Changes the specified attributes for the default broker or a broker at the specified host and port.

`metrics bkr [-b` *hostName:port*`]`
`[-m` *metricType*`]`
`[-int` *interval*`]`
	Displays broker metrics for the default broker or a broker at the specified host and port.
	Use the -m option to specify the type of metric to display:
	`ttl` Total of messages in and out of the broker (default).
	`rts` Provides the same information as ttl, but specifies the number of messages per second.
	`cxn` Connections, virtual memory heap, threads
	Use the -int option to specify the interval (in seconds) at which to display the metrics. The default is 5 seconds

Remember that you must specify the broker host name and port number when using any of the subcommands listed in Table 6-4 unless you are targeting the broker running on `localhost` at port 7676

Querying a Broker

To query and display information about a single broker, use the query subcommand. For example,

```
imqcmd query bkr
```

This produces output like the following:

```
%imqcmd query bkr

Querying the broker specified by:

-------------------------
Host         Primary Port
-------------------------
localhost    7676

Auto Create Queues                     true
Auto Create Topics                     true
Auto Created Queue Delivery Policy     Round Robin
Cluster Broker List (active)           myhost/192.18.116.221:7676
Cluster Broker List (configured)
Cluster Master Broker
Cluster URL
Current Number of Messages in System   0
Current Size of Messages in System     0
Instance Name                          imqbroker
Log Level                              INFO
Log Rollover Interval (seconds)        604800
Log Rollover Size (bytes)              0 (unlimited)
Max Message Size                       70m
Max Number of Messages in System       0 (unlimited)
Max Size of Messages in System         0 (unlimited)
Primary Port                           7676
Version                                3.0

Successfully queried the broker.
```

Updating a Broker

You can use the `update` subcommand to update any of the broker properties listed in Table 6-5. Note that updates to the broker are automatically written to the broker's instance configuration file.

Table 6-5 Broker Properties

Properties	Description
`imq.autocreate.queue`	Specifies whether a broker is allowed to auto-create a queue destination. `True` by default.
`imq.autocreate.topic`	Specifies whether a broker is allowed to auto-create a topic destination. `True` by default.
`imq.queue.deliverypolicy`	Specifies the default delivery policy of auto-created queues. Values are: `s` (single), `r` (round-robin), or `f` (failover). Default is `s`.
`imq.cluster.url`	Specifies the location of the cluster configuration file. For more information, see"Cluster Configuration Properties" on page 126.
`imq.log.level`	Specifies the log level as one of the following: `NONE`, `ERROR`, `WARNING`, `INFO`. Default is `INFO`.
`imq.log.file.rolloversecs`	The age (in seconds) before the log file is rolled over. A value of 0 means no rollover based on the age of the file. Default is 604800 (7 days).
`imq.log.file.rolloverbytes`	Specifies the maximum size of the log file before it is rolled over. A value of 0 means no rollover based on file size. Default is 0.
`imq.message.max_size`	Specifies the maximum size of a message in bytes. Default is `70m`.
`imq.system.max_count`	Specifies the maximum number of messages in memory and disk. A value of `0` means no limit. Default is `0`.
`imq.system.max_size`	Specifies the maximum total size of messages in memory and disk. A value of 0 means no limit. Default is `0`.
`imq.portmapper.port`	Specifies the number of the port mapper port. Default is `7676`.

For example, the following command changes the default delivery policy for queues from single to round-robin.

```
imqcmd update bkr -o “imq.queue.deliverypolicy=r”
```

Managing Connection Services

The Command utility includes a number of subcommands that allows you to do the following

- list available connection services
- display information about a particular service
- update the attributes of a service
- pause and resume services

For an overview of MQ connection services, see “Connection Services” on page 46.

Table 6-6 lists the `imqcmd` subcommands that control connection services. If no host name or port is specified, they are assumed to be `localhost, 7676`.

Table 6-6 `imqcmd` Subcommands Used to Manage Connection Services

Subcommand Syntax	Description
`list svc [-b` *hostName:port*`]`	Lists all connection services on the default broker or on a broker at the specified host and port.

`metrics svc -n` *serviceName*
`[-b` *hostName:port*`]`
`[-m` *metricType*`] [-int` *interval*`]`
	Lists metrics for the specified service on the default broker or on a broker at the specified host and port.
	Use the -m option to specify the type of metric to display:
	`ttl` Total of messages in and out of the broker (default).
	`rts` Provides the same information as ttl, but specifies the number of messages per second.
	`cxn` Connections, virtual memory heap, threads
	Use the -int option to specify the interval (in seconds) at which to display the metrics. The default is 5 seconds.

Table 6-6 `imqcmd` Subcommands Used to Manage Connection Services *(Continued)*

Subcommand Syntax	Description
`query svc -n` *serviceName* `[-b` *hostName:port*`]`	Displays information about the specified service running on the default broker or on a broker at the specified host and port.
`pause svc -n` *serviceName* `[-b` *hostName:port*`]`	Pauses the specified service running on the default broker or on a broker at the specified host and port. You cannot pause the admin service.
`resume svc -n` *serviceName* `[-b` *hostName:port*`]`	Resumes the specified service running on the default broker or on a broker at the specified host and port.

`update svc -n` *serviceName*
`[-b` *hostName:port*`]`
`-o` *attribute=value*
`[-o` *attribute=value1*`]...`
	Updates the specified attribute of the specified service running on the default broker or on a broker at the specified host and port. For a description of service attributes, see Table 6-8 on page 151.

A broker supports communication with both application clients and administration clients. The connection services currently available from an MQ broker are shown in Table 6-7 on page 149. The values in the Service Name column are the values you use to specify a service name for the -n option. (As shown in the table, each service is specified by the service type it uses—`NORMAL` (JMS) or `ADMIN`—and an underlying transport layer.)

Table 6-7 Connection Services Supported by a Broker

Service Name	Service Type	Protocol Type
jms	NORMAL (JMS message delivery)	tcp
ssljms (Enterprise Edition)	NORMAL (JMS message delivery)	tls (SSL-based security)
httpjms (Enterprise Edition)	NORMAL (JMS message delivery)	http
httpsjms (Enterprise Edition)	NORMAL (JMS message delivery)	https (SSL-based security)
admin	ADMIN	tcp
ssladmin (Enterprise Edition)	ADMIN	tls (SSL-based security)

Listing Connection Services

To list available connection services on a broker, use a command like the following:

```
imqcmd list svc [-b hostName:portNumber]
```

For example, the following command lists the services available for the broker running on the host MyServer on port 6565.

```
imqcmd list svc -b MyServer:6565
```

The following command lists all services on the broker running on `localhost` at port 7676:

```
imqcmd list svc
```

The command will output information like the following:

```
Listing all the services on the broker specified by:

Host                 Primary Port
localhost            7676

Service Name     Port Number       Service State
admin            33984 (dynamic)   RUNNING
httpjms                            UNKNOWN
httpsjms         -                 UNKNOWN
jms              33983 (dynamic)   RUNNING
ssladmin         dynamic           UNKNOWN
ssljms           dynamic           UNKNOWN

Successfully listed services.
```

Querying and Updating Service Properties

To query and display information about a single service, use the query subcommand. For example,

```
imqcmd query svc -n jms
```

This produces output like the following:

```
Querying the service where:

Service Name
jms

On the broker specified by:

Host                         Primary Port
localhost                    7676


Current Number of Allocated Threads    120
Current Number of Connections          20
Max Number of Threads                  1000
Min Number of Threads                  50
Port Number                            42019 (dynamic)
Service Name                           jms
Service State                          RUNNING

Successfully queried the service.
```

You can use the `update` subcommand to change the value of one or more of the service attributes listed in Table 6-8.

Table 6-8 Connection Service Attributes

Attribute	Description
`port`	The port assigned to the service to be updated (does not apply to httpjms or httpsjms).
`minThreads`	The minimum number of threads assigned to the service.
`maxThreads`	The maximum number of threads assigned to the service.

The following command changes the minimum number of threads assigned to the jms service to 20.

```
imqcmd update svc -n jms -o “minThreads=20”
```

Pausing and Resuming a Service

To pause any service other than the admin service (which cannot be paused), use a command like the following:

```
imqcmd pause svc -n serviceName
```

To resume a service, use a command like the following:

```
imqcmd resume svc -n serviceName
```

Managing Destinations

All MQ messages are routed to their consumer clients by way of destinations, queues and topics, created on a particular broker. You are responsible for managing these destinations on the broker. This involves using the Command utility to create and destroy destinations, to list destinations, to display information about destinations, and to purge messages. For an introduction to destinations, see "Physical Destinations" on page 65.

Table 6-9 provides a summary of the `imqcmd` destination subcommands. Remember to specify the host name and port of the broker if this is not the default (`localhost:7676`) broker.

Table 6-9 `imqcmd` Subcommands Used to Manage Destinations

Subcommand	Description
`list dst [-tmp]`	Lists all destinations, with option of listing temporary destinations as well (see "Temporary Destinations" on page 68).

`create dst -t` *type*
`-n` *destName*
`[-o` *attribute=value*`]`
`[-o` *attribute=value*`1]...`
	Creates a destination of the specified type, with the specified name, and the specified attributes. Destination names must contain only alphanumeric characters (no spaces) and can begin with an alphabetic character or the character "_"

`destroy dst -t` *type* `-n` *destName*	Destroys the destination of the specified type and name.
`purge dst -t` *type* `-n` *destName*	Purges messages at the destination with the specified type and name.
`query dst -t` *type* `-n` *destName*	Lists information about the destination of the specified type and name.

Table 6-9 `imqcmd` Subcommands *(Continued)*Used to Manage Destinations *(Continued)*

Subcommand	Description
`update dst -t` *type* `-n` *destName* `-o` *attribute=value* `[-o` *attribute=value1*`]...`	Updates the value of the specified attributes at the specified destination. The attribute name may be any of the attributes described in Table 6-10.

Creating Destinations

When creating a destination, you must specify its type (topic or queue) and, if needed, specify values for the destination's attributes. Default values for these attributes are set in the broker's configuration file (see "Configuration Files" on page 114.)

Destroying a destination purges all messages at that destination and removes it from the broker; the operation is not reversible.

Table 6-10 describes the attributes that can be set for each type of destination when you create the destination.

Table 6-10 Destination Attributes

Destination Type	Attribute	Default Value	Description

Queue	`queueDeliveryPolicy`	`Single`	Describes the algorithm used to route messages.
			Values are
			f = Failover
			r = Round robin
			s = Single

Queue	`maxTotalMsgBytes`	`0` (unlimited)	Maximum total size in bytes of messages allowed in the queue.
Queue	`maxNumMsgs`	`0` (unlimited)	Maximum number of messages allowed in the queue
Queue	`maxBytesPerMsg`	`0` (unlimited)	Maximum size of any single message allowed in the queue.
Topic	`maxBytesPerMsg`	`0` (unlimited)	Maximum size of any single message posted to the topic.

- To create a queue destination, enter a command like the following:

  ```
  imqcmd create dst -n myQueue -t q -o “queueDeliveryPolicy=f”
  ```

  Note that a destination name must be a valid Java identifier.

- To create a topic destination, enter a command like the following:

  ```
  imqcmd create dst -n myTopic -t t -o “maxBytesPerMsg=5000”
  ```

Getting Information About Destinations

To get information about the current value of a destination’s attributes, use a command like the following:

```
imqcmd query dst -t q -n XQueue
```

You can then use the `update imqcmd` subcommand to change the values of one or more attributes.

To list all destinations on a particular broker, say the broker running on `myHost` at port 4545, use a command like the following:

```
imqcmd list dst -b myHost:4545
```

The `list` command can optionally include temporary destinations (using the `-tmp` option). These are destinations created by client applications that need a destination at which to receive replies to messages sent to other clients (see “Temporary Destinations” on page 68). You cannot destroy these destinations; they can only be destroyed by API calls made by the client application when there are no more active message consumers.

Updating Destinations

You can change the attributes of a destination by using the `update dst` subcommand and the -o option to specify the attribute to update. You can use the `-o` option more than once if you want to update more than one attribute. For example, the following command changes the `maxBytesPerMsg` attribute to `1000` and the `MaxNumMsgs` to `2000`:

```
imqcmd update dst -t q -n myQueue -o “maxBytesPerMsg=1000”
             -o maxNumMsgs=2000
```

See Table 6-10 on page 153 for a list of the attributes that you can update.

You cannot use the `update dst` subcommand to update the *type* of a destination or to update the queue delivery policy for a queue.

Purging Destinations

You can purge all messages currently queued at a destination. Purging a destination means that all messages queued at the physical destination are deleted. You might want to purge messages when the messages accumulated at a destination are taking up too much of the system's resources. This might happen when a queue does not have any registered consumer clients and is receiving many messages. It might also happen if inactive durable subscribers to a topic do not become active. In both cases, messages are held unnecessarily.

To purge messages at a destination, enter commands like the following:

```
imqcmd purge dst -n myQueue -t q
```

```
imqcmd purge dst -n myTopic -t t
```

In the case where you have shut down the broker and do not want old messages to be delivered when you restart it, use the `reset` subcommand of the `imqbrokerd` command to purge stale messages; for example:

```
imqbrokerd -reset messages
```

This saves you the trouble of purging destinations after restarting the broker.

Destroying Destinations

To destroy a destination, enter a command like the following:

```
imqcmd destroy dst -t q -n myQueue
```

Managing Durable Subscriptions

You might need to use `imqcmd` subcommands to manage a broker's durable subscriptions. A *durable subscription* is a subscription to a topic that is registered by a client as durable; it has a unique identity and it requires the broker to retain messages for that subscription even when its consumer becomes inactive. Normally, the broker may only delete a message held for a durable subscriber when the message expires.

Table 6-12 provides a summary of the `imqcmd` durable subscription subcommands. Remember to specify the host name and port of the broker if this is not the default (`localhost:7676`) broker.

Table 6-11 `imqcmd` Subcommands Used to Manage Durable Subscriptions

Subcommand	Description
`list dur -d` *destination*	Lists all durable subscriptions for the specified destination.
`destroy dur -n` *subscrName* `-c` *client_id*	Destroys the specified durable subscription for the specified Client Identifier (see "Client Identifiers" on page 38).
`purge dur -n` *subscrName* `-c` *client_id*	Purges all messages for the specified Client Identifier (see "Client Identifiers" on page 38).

For example, the following command lists all durable subscriptions to the topic `SPQuotes`

```
imqcmd list dur -d SPQuotes
```

For each durable subscription to a topic, the `list` subcommand returns the name of the durable subscription, the client ID of the user, the number of messages queued to this topic, and the state of the durable subscription (active/inactive). For example:

```
Listing all the durable subscriptions on the topic myTopic
on the broker specified by:
-------------------------
Host         Primary Port
-------------------------
localhost    7676

Name        Client ID       Number of   Durable Sub
                            Messages      State
----------------------------------------------------------------
myDurable   myClientID       1            INACTIVE

Successfully listed durable subscriptions.
```

You can use the information returned from the `list` command to identify a durable subscription you might want to destroy or for which you want to purge messages. Use the name of the subscription and the client ID to identify the subscription. For example:

```
imqcmd destroy dur -n myDurable -c myClientID
```

Managing Transactions

All transactions initiated by client applications are tracked by the broker. These can be simple MQ transactions or distributed transactions managed by an XA resource manager (see "Local Transactions" on page 40). All transaction have an MQ transaction ID—a 64 bit number that uniquely identifies a transaction on the broker. Distributed transactions also have a distributed transaction ID (XID) assigned by the distributed transaction manager—up to 128 bytes long. MQ maintains the association of an MQ transaction ID with an XID.

For distributed transactions, in cases of failure, it is possible that transactions could be left in a `PREPARED` state without ever being committed. Hence, as an administrator you might need to monitor and then roll back or commit transactions left in a prepared state.

Table 6-12 provides a summary of the `imqcmd` transactions subcommands. Remember to specify the host name and port of the broker if this is not the default (`localhost:7676`) broker.

Table 6-12 `imqcmd` Subcommands Used to Manage Transactions

Subcommand	**Description**
`list txn`	Lists all transactions, being tracked by the broker.
`query txn -n` *transaction_id*	Lists information about the specified transaction.
`commit txn -n` *transaction_id*	Commits the specified transaction.
`rollback txn -n` *transaction_id*	Rolls back the specified transaction.

For example, the following command lists all transactions in a broker.

```
imqcmd list txn
```

For each transaction, the `list` subcommand returns the transaction ID, state, user name, number of messages or acknowledgements, and creation time. For example:

```
Listing all the transactions on the broker specified by:
-------------------------
Host         Primary Port
-------------------------
localhost    7676

---------------------------------------------------------------
Transaction ID  State    User name    # Msgs/   Creation time
                                      # Acks
---------------------------------------------------------------

64248349708800  PREPARED  guest      4/0      1/30/02 10:08:31 AM
64248371287808  PREPARED  guest      0/4      1/30/02 10:09:55 AM

Successfully listed transactions.
```

The command shows all transactions in the broker, both local and distributed. You can only commit or roll back transactions in the `PREPARED` state. You should only do so if you know that the transaction has been left in this state by a failure and is not in the process of being committed by the distributed transaction manager.

For example, if the broker's auto-rollback property is set to false (see Table 2-4 on page 53), then you have to manually commit or roll back transactions found in a `PREPARED` state at broker startup.

The `list` subcommand also shows the number of messages that were produced in the transaction and the number of messages that were acknowledged in the transaction (`#Msgs/#Acks`). These messages will not be delivered and the acknowledgements will not be processed until the transaction is committed.

The `query` subcommand lets you see the same information plus a number of additional values: the Client ID, connection identification, and distributed transaction ID (XID). For example,

```
imqcmd query txn -n 64248349708800
```

produces the following output:

```
Querying the transaction where:
-------------------------
Transaction ID
-------------------------
64248349708800

On the broker specified by:

-------------------------
Host         Primary Port
-------------------------
localhost    7676


Client ID
Connection                 guest@192.18.116.219:62209->jms:62195
Creation time              1/30/02 10:08:31 AM
Number of acknowledgements 0
Number of messages         4
State                      PREPARED
Transaction ID             64248349708800
User name                  guest
XID
6469706F6C7369646577696E64657231303132343134313130303373230

Successfully queried the transaction.
```

The `commit` and `rollback` subcommands can be used to commit or roll back a distributed transaction. As mentioned previously, only a transaction in the `PREPARED` state can be committed or rolled back. For example:

```
imqcmd commit txn -n 64248349708800
```

It is also possible to configure the broker to automatically roll back transactions in the `PREPARED` state at broker startup. See the `imq.transaction.autorollback` property in Table 2-4 on page 53 for more information.

Chapter 7

Managing Administered Objects

The use of administered objects enables the development of client applications that are portable to other JMS providers. *Administered objects* are objects that encapsulate provider-specific configuration and naming information. These objects are normally created by a Sun™ ONE Message Queue (MQ) administrator and used by client applications to obtain connections to the broker, which are then used to send messages to and receive messages from physical destinations.

For an overview of administered objects, see“MQ Administered Objects” on page 75.

MQ provides two administration tools for creating and managing administered objects: the command line Object Manager utility (`imqobjmgr`) and the GUI Administration Console. These tools enable you to do the following:

- Add or delete administered objects to an object store.
- List existing administered objects.
- Query and display information about an administered object.
- Modify an existing administered object in the object store.

This chapter explains how you use the Object Manager utility (`imqobjmgr`) to perform these tasks. For information about the Administration Console, see Chapter 4, "Administration Console Tutorial."

About Object Stores

Administered objects are placed in a readily available object store where they can be accessed by client applications through a JNDI lookup. There are two types of object stores you can use: a standard LDAP directory server or a file-system object store.

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

File-system Store MQ also supports a file-system object store implementation. While the file-system object store is not fully tested and is therefore not recommended for production systems, it has the advantage of being very easy to use in development environments. Rather than setting up an LDAP server, all you have to do is create a directory on your local file system. Any user with access to that directory can use MQ administration tools to create and manage administered objects.

Administered Objects

For an overview of administered objects, see "MQ Administered Objects" on page 75.

MQ administered objects are of two basic kinds: connection factories and destinations. *Connection factory* administered objects are used by client applications to create a connection to a broker. *Destination* administered objects are used by client applications to identify the destination to which a producer is sending messages or from which a consumer is retrieving messages. (A special *SOAP endpoint* administered object is used for SOAP messaging—see the MQ *Developer's Guide* for more information.)

Depending on the message delivery model (point-to-point or publish/subscribe), connection factories and destinations of a specific type can be used. In point-to-point programming, for example, a queueConnectionFactory and a queue destination can be used. Similarly, in publish and subscribe programming, a topicConnectionFactory and a topic destination can be used. Non-specific connection factory and destination administered object types are also available, as are connection factory types that support distributed transactions (see Table 1-1 on page 38 for all the supported types).

Object Manager Utility (imqobjmgr)

The Object Manager utility allows you to create and manage MQ administered objects. This section describes the basic `imqobjmgr` command syntax, provides a listing of subcommands, and summarizes `imqobjmgr` command options. Subsequent sections explain how you use the `imqobjmgr` subcommands to accomplish specific tasks.

Syntax of Command

The general syntax of the `imqcmd` command is as follows:

```
imqobjmgr subcommand [options]
imqobjmgr -h|H
imqobjmgr -v
```

Note that if you specify the -v, -h, or -H options, no subcommands specified on the command line are executed. For example, if you enter the following command, version information is displayed but the `list` subcommand is not executed.

```
imqobjmgr list -v
```

imqobjmgr Subcommands

The Object Manager utility (`imqobjmgr`) includes the subcommands listed in Table 7-1:

Table 7-1 `imqobjmgr` Subcommands

Subcommand	Description
`add`	Adds an administered object to the object store.
`delete`	Deletes an administered object from the object store.
`list`	Lists administered objects in the object store.
`query`	Displays information about the specified administered object.
`update`	Modifies an existing administered object in the object store.

Summary of imqobjmgr Command Options

Table 7-2 lists the options to the `imqobjmgr` command. For a discussion of their use, see the task-based sections that follow.

Table 7-2 `imqobjmgr` Options

Option	Description
`-f`	Performs action without user confirmation.
`-h`	Displays usage help. Nothing else on the command line is executed.
`-H`	Displays usage help, attribute list, and examples. Nothing else on the command line is executed.
`-i` *fileName*	Specifies the name of an input file containing all or part of the subcommand clause, specifying (object type, lookup name, object attributes, object store attributes, or other options). Typically used for repetitive information, such as object store attributes.
`-j` *attribute=value*	Specifies attributes necessary to identify and access a JNDI object store.
`-javahome`	Specifies an alternate Java 2 compatible runtime to use (default is to use the runtime bundled with the product).
`-l` *lookupName*	Specifies the JNDI lookup name of an administered object. This name must be unique in the object store's context.
`-o` *attribute=value*	Specifies attributes of an administered object.
`-pre`	Preview mode. Indicates what will be done without performing the command.
`-r` *read-only_state*	Specifies whether an administered object is a read-only object. A value of `true` indicates the administered object is a read-only object. JMS clients cannot modify the attributes of read-only administered objects. The read-onlystate is set to `false` by default.
`-s`	Silent mode. No output will be displayed.

Table 7-2 `imqobjmgr` Options *(Continued)*

Option	Description

`-t` *type*	Specifies the type of an MQ administered object:
	q = queue
	t = topic
	cf = ConnectionFactory
	qf = queueConnectionFactory
	tf = topicConnectionFactory
	xcf = XA ConnectionFactory (distributed transactions)
	xqf = XA queueConnectionFactory (distributed transactions)
	xtf = XA topicConnectionFactory (distributed transactions)
	e = SOAP endpoint[1]

`-v`	Displays version information. Nothing else on the command line is executed.

[1] This administered object type is used to support SOAP messages (see the MQ *Developer's Guide*).

The following section describes information that you need to provide when working with any `imqobjmgr` subcommand.

Required Information

When performing most tasks related to administered objects, the administrator must specify the following information as options to `imqobjmgr` subcommands:

- The type of the administered object:

  The allowed types are shown in Table 7-2.

- The JNDI lookup name of the administered object:

  This is the logical name that will be used in the client code to refer to the administered object (using JNDI) in the object store.

- Attributes of the administered object:

  - For queues and topics: The name of the physical destination on the broker. This is the name that was specified with the `-n` option to the `imqcmd create` subcommand. If you do not specify the name, the default name of `Untitled_Destination_Object` will be used.
  - For connection factories: The host name and port number of the broker to which the client will connect. If you do not specify this information, the local host and default port number (7676) are used. The section "Administered Object Attributes" on page 167 explains how you specify object attributes.

  For additional attributes, see "Administered Object Attributes" on page 167.

- Attributes of the JNDI object store:

  This information depends on whether you are using a file-system store or LDAP server, but must include the following attributes:

  - The type of JNDI implementation (initial context attribute). For example, file-system or LDAP.
  - The location of the administered object in the object store (provider URL attribute), that is, its "folder" as it were.
  - The user name, password, and authorization type, if any, required to access the object store.

  For more information about object store attributes see "Object Store Attributes" on page 169.

Administered Object Attributes

The attributes of an administered object are specified using attribute-value pairs. The following sections describe these attributes.

Connection Factory Administered Objects

Connection factory (and XA connection factory) administered objects have the attributes listed in Table 7-3. The two attributes you are primarily concerned with are `imqBrokerHostPort` and `imqBrokerHostName`, which you use to specify the broker to which the client application will establish a connection. The section, "Adding a Connection Factory" on page 174, explains how you specify these attributes when you add a connection factory administered object to your object store.

For more descriptions of connection factory attributes and information on how they are used, see the MQ *Developer's Guide* and the JavaDoc API documentation for the MQ class `com.sun.messaging.ConnectionConfiguration`.

Table 7-3 Connection Factory Attributes

Attribute/property name	Type	Default Value
imqAckOnAcknowledge	String	not specified
imqAckOnProduce	String	not specified
imqAckTimeout	String	0 milliseconds
imqBrokerHostName	String	localhost
imqBrokerHostPort	String	7676
imqBrokerServicePort	String	0
imqConfiguredClientID	String	not specified
imqConnectionType	String	TCP
imqConnectionURL	String	http://localhost/imq/tunnel
imqDefaultPassword	String	guest
imqDefaultUsername	String	guest
imqDisableSetClientID	String	false
imqFlowControlCount	String	100
imqFlowControlIsLimited	String	false
imqFlowControlLimit	String	1000

Table 7-3 Connection Factory Attributes *(Continued)*

Attribute/property name	Type	Default Value
imqLoadMaxToServerSession	String	false
imqQueueBrowserMaxMessagesPerRetrieve	String	1000
imqQueueBrowserRetrieveTimeout	String	60,000 milliseconds
imqReconnect	Boolean	false
imqReconnectDelay	String	30,000 milliseconds
imqReconnectRetries	String	0
imqSetJMSXAppID	String	false
imqSetJMSXConsumerTXID	String	false
imqSetJMSXProducerTXID	String	false
imqSetJMSXRcvTimestamp	String	false
imqSetJMSXUserID	String	false
imqSSLIsHostTrusted	String	true
imqJMSDeliveryMode	Integer	2 (persistent)
imqJMSExpiration	Integer	0 (does not expire)
imqJMSPriority	Integer	4 (normal priority)
imqOverrideJMSDeliveryMode	Boolean	false
imqOverrideJMSExpiration	Boolean	false
imqOverrideJMSPriority	Boolean	false
imqOverrideJMSHeadersToTemporaryDestinations	Boolean	false

Destination Administered Objects

The destination administered object that identifies a physical topic or queue destination has the attributes listed in Table 7-4. The section, "Adding a Topic or Queue" on page 175, explains how you specify these attributes when you add a destination administered object to your object store.

The attribute you are primarily concerned with is `imqDestinationName`. This is the name you assign to the physical destination that corresponds to the topic or queue administered object. You can also provide a description of the destination that will help you distinguish it from others that you might create to support many applications.

For more information, see the JavaDoc API documentation for the MQ class `com.sun.messaging.DestinationConfiguration`.

Table 7-4 Destination Attributes

Attribute/property name	Type	Default
`imqDestinationDescription`	String	`A Description for the destination Object`
`imqDestinationName`	String[1]	`Untitled_Destination_Object`

[1] Destination names can contain only alphanumeric characters (no spaces) and must begin with an alphabetic character or the characters "_" and/or "$".

Object Store Attributes

The attributes of the object store are specified using the `-j` option and consist of attribute-value pairs. In general, you must specify the following attributes:

Initial Context and Location Information

The format for these entries differs depending on whether you are using a file-system store or LDAP server.

File-system store As an example of using a file-system store, create a folder called *MyObjstore* on the C drive, and specify the following values for the initial context and location attributes, respectively:

```
-j "java.naming.factory.initial=
          com.sun.jndi.fscontext.RefFSContextFactory"
```

```
-j "java.naming.provider.url=file:///C:/MyObjStore"
```

LDAP server As an example of using an LDAP server, specify the following values for the initial context and location attributes, respectively:

```
-j "java.naming.factory.initial=
          com.sun.jndi.ldap.LdapCtxFactory"
```

```
-j "java.naming.provider.url=ldap://mydomain.com:389/o=imq"
```

Security Information (LDAP Only)

The format for these entries differs depending on the LDAP provider. You should also consult the documentation provided with your LDAP implementation to determine whether security information is required on all operations or only on operations that change the stored data.

Security attributes look like this:

```
-j "java.naming.security.principal=
      uid=fooUser, ou=People, o=imq"
-j "java.naming.security.credentials=fooPasswd"
-j "java.naming.security.authentication=simple"
```

Table 7-5 describes these entries:

Table 7-5 Security Attributes for the Object Store

Attribute	Description
…`principal`	The identity of the principal for authenticating the caller to the service.The format of this entry depends on the authentication scheme. If this property is unspecified, the behavior is determined by the service provider.
…`credentials`	The credentials of the principal for authenticating the caller to the service. The value of the property depends on the authentication scheme. For example, it could be a hashed password, clear-text password, key, certificate, and so on. If this property is unspecified, the behavior is determined by the service provider.
…`authentication`	Security level to use. Its value is one of the following key words: `none`, `simple`, `strong`. If this property is unspecified, the behavior is determined by the service provider. If you specify `simple, imqobjmgr` will prompt for any missing principal or credential values. This will allow you a more secure way of providing identifying information.

Using Input Files

The imqobjmgr command allows you to specify the name of an input file that uses java property file syntax to represent all or part of the imqobjmgr subcommand clause.

Using an input file with the Object Manager utility (imqobjmgr) is especially useful to specify object store attributes, which are likely to be the same across multiple invocations of imqobjmgr and which normally require a lot of typing. Using an input file can also allow you to avoid a situation in which you might otherwise exceed the maximum number of characters allowed for the command line.

The general syntax for a imqobjmgr input file is as follows (the version property applies only to the input file—it is not a command line option—and its value must be set to 2.0):

```
version=2.0
cmdtype=[ add | delete | list | query | update ]
obj.type=[ q | t | qf | tf | cf | xqf | xtf | xcf | e ]
obj.lookupName=lookup name
obj.attrs.objAttrName1=value1
obj.attrs.objAttrName2=value2
obj.attrs.objAttrNameN=valueN
...
objstore.attrs.objStoreAttrName1=value1
objstore.attrs.objStoreAttrName2=value2
objstore.attrs.objStoreAttrNameN=valueN
...
```

As an example of how you can use an input file, consider the following `imqobjmgr` command:

```
imqobjmgr add
       -t qf
       -l "cn=myQCF"
       -o "imqBrokerHostName=foo"
       -o "imqBrokerHostPort=777"
       -j "java.naming.factory.initial=
                com.sun.jndi.ldap.LdapCtxFactory"
       -j "java.naming.provider.url=
                ldap://mydomain.com:389/o=imq"
       -j "java.naming.security.principal=
                uid=fooUser, ou=People, o=imq"
       -j "java.naming.security.credentials=fooPasswd"
       -j "java.naming.security.authentication=simple"
```

This command can be encapsulated in a file, say `MyCmdFile`, that has the following contents:

```
    version=2.0
    cmdtype=add
    obj.type=qf
    obj.lookupName=cn=myQCF
    obj.attrs.imqBrokerHostName=foo
    obj.attrs.imqBrokerHostPort=777
    objstore.attrs.java.naming.factory.initial=\
              com.sun.jndi.ldap.LdapCtxFactory
    objstore.attrs.java.naming.provider.url=\
              ldap://mydomain.com:389/o=imq
    objstore.attrs.java.naming.security.principal=\
              uid=fooUser, ou=People, o=imq
    objstore.attrs.java.naming.security.credentials=fooPasswd
    objstore.attrs.java.naming.security.authentication=simple
```

You can then use the -i option to pass this file to the Object Manager utility (imqobjmgr):

```
imqobjmgr -i MyCmdFile
```

You can also use the input file to specify some options, while using the command line to specify others. This allows you to use the input file to specify parts of the subcommand clause that is the same across many invocations of the utility. For example, the following command specifies all the options needed to add a connection factory administered object, except for those that specify where the administered object is to be stored.

```
imqobjmgr add
        -t qf
        -l "cn=myQCF"
        -o "imqBrokerHostName=foo"
        -o "imqBrokerHostPort=777"
        -i MyCmdFile
```

In this case, the file MyCmdFile would contain the following definitions:

```
    version=2.0
    objstore.attrs.java.naming.factory.initial=\
              com.sun.jndi.ldap.LdapCtxFactory
    objstore.attrs.java.naming.provider.url=\
              ldap://mydomain.com:389/o=imq
    objstore.attrs.java.naming.security.principal=\
              uid=fooUser, ou=People, o=imq
    objstore.attrs.java.naming.security.credentials=fooPasswd
    objstore.attrs.java.naming.security.authentication=simple
```

Additional examples of input files can be found at the following location:

```
IMQ_HOME/demo/imqobjmgr
```

Adding and Deleting Administered Objects

This section explains how you add administered objects for connection factories and topic or queue destinations to the object store.

NOTE The Object Manager utility (`imqobjmgr`) lists and displays only MQ administered objects. If an object store should contain a non-MQ object with the same lookup name as an administered object that you wish to add, you will receive an error when you attempt the add operation.

Adding a Connection Factory

To enable client applications to obtain a connection to the broker, you add an administered object that represents the type of connections the client applications want: a topic connection factory or a queue connection factory

To add a queue connection factory, use a command like the following:

```
imqobjmgr add
  -t qf
  -l "cn=myQCF"
  -o "imqBrokerHostName=myHost"
  -o "imqBrokerHostPort=7272"
  -j "java.naming.factoryinitial=
              com.sun.jndi.ldap.LdapCtxFactory"
  -j "java.naming.provider.url=ldap://mydomain.com:389/o=imq"
  -j "java.naming.security.principal=
        uid=fooUser, ou=People, o=imq"
  -j "java.naming.security.credentials=fooPasswd"
  -j "java.naming.security.authentication=simple"
```

The preceding command creates an administered object whose lookup name is cn=myQCF and which connects to a broker running on myHost and listens on port 7272. The administered object is stored in an LDAP server.

NOTE If you are using an LDAP server to store the administered object, it is important that you assign a lookup name that has the prefix “cn=” as in the example above. You specify the lookup name with the -l option. You do not have to use this prefix if you are using a file-system object store.

You can accomplish the same thing by specifying an input file as an argument to the imqobjmgr command. For more information, see “Using Input Files” on page 171.

Adding a Topic or Queue

To enable client applications to access physical destinations on the broker, you add administered objects that identify these destinations, to the object store.

It is best to first create the physical destinations before adding the corresponding administered objects to the object store. Use the Command utility (imqcmd) to create the physical destinations on the broker that are identified by destination administered objects in the object store. For information about creating physical destinations, see “Managing Destinations” on page 152.

The following command adds an administered object that identifies a topic destination whose lookup name is myTopic and whose physical destination name is TestTopic. The administered object is stored in an LDAP server.

```
imqobjmgr add
-t t
-l "cn=myTopic"
-o "imqDestinationName=TestTopic"
-j "java.naming.factory.initial=
       com.sun.jndi.ldap.LdapCtxFactory"
-j "java.naming.provider.url=
       ldap://mydomain.com:389/o=imq"
-j "java.naming.security.principal=
       uid=fooUser, ou=People, o=imq"
-j "java.naming.security.credentials=fooPasswd"
-j "java.naming.security.authentication=simple"
```

This is the same command, only the administered object is stored in a Solaris file system:

```
imqobjmgr add
      -t t
      -l "cn=myTopic"
      -o "imqDestinationName=TestTopic"
      -j "java.naming.factory.initial=
              com.sun.jndi.fscontext.RefFSContextFactory"
      -j "java.naming.provider.url=
              file:///home/foo/imq_admin_objects"
```

In the LDAP server case, as an example, you could use an input file, `MyCmdFile`, to specify the subcommand clause. The file would contain the following text:

```
    version=2.0
    cmdtype=add
    obj.type=t
    obj.lookupName=cn=myTopic
    obj.attrs.imqDestinationName=TestTopic
    objstore.attrs.java.naming.factory.initial=
            com.sun.jndi.fscontext.RefFSContextFactory
    objstore.attrs.java.naming.provider.url=
            file:///home/foo/imq_admin_objects
    objstore.attrs.java.naming.security.principal=
            uid=fooUser, ou=People, o=imq
    objstore.attrs.java.naming.security.credentials=fooPasswd
    objstore.attrs.java.naming.security.authentication=simple
```

Use the -i option to pass the file to the `imqobjmgr` command:

```
imqobjmgr -i MyCmdFile
```

NOTE If you are using an LDAP server to store the administered object, it is important that you assign a lookup name that has the prefix “cn=” as in the example above. You specify the lookup name with the `-l` option. You do not have to use this prefix if you are using a file-system object store.

Adding a queue object is exactly the same, except that you specify q for the -t option.

Deleting Administered Objects

Use the delete subcommand to delete an administered object. You must specify the lookup name of the object, its type, and its location.

The following command deletes an administered object for a topic whose lookup name is cn=myTopic and which is stored on an LDAP server.

```
imqobjmgr delete
     -t t
     -l "cn=myTopic"
     -j "java.naming.factory.initial=
            com.sun.jndi.ldap.LdapCtxFactory"
     -j "java.naming.provider.url=
            ldap://mydomain.com:389/o=imq"
     -j "java.naming.security.principal=
            uid=fooUser, ou=People, o=imq"
     -j "java.naming.security.credentials=fooPasswd"
     -j "java.naming.security.authentication=simple"
```

Getting Information

Use the list and query subcommands to list administered objects in the object store and to display information about an individual object.

Listing Administered Objects

Use the list subcommand to get a list of all administered objects or to get a list of all administered objects of a specific type. The following sample code assumes that the administered objects are stored in an LDAP server.

The following command lists all objects.

```
imqobjmgr list
    -j "java.naming.factory.initial=
            com.sun.jndi.ldap.LdapCtxFactory"
    -j "java.naming.provider.url=
            ldap://mydomain.com:389/o=imq"
    -j "java.naming.security.principal=
            uid=fooUser, ou=People, o=imq"
    -j "java.naming.security.credentials=fooPasswd"
    -j "java.naming.security.authentication=simple"
```

The following command lists all objects of type queue.

```
imqobjmgr list
  -t q
  -j "java.naming.factory.initial=
            com.sun.jndi.ldap.LdapCtxFactory"
  -j "java.naming.provider.url=
            ldap://mydomain.com:389/o=imq"
  -j "java.naming.security.principal=
            uid=fooUser, ou=People, o=imq"
  -j "java.naming.security.credentials=fooPasswd"
  -j "java.naming.security.authentication=simple"
```

Information About a Single Object

Use the query subcommand to get information about an administered object. You must specify the object's lookup name and the attributes of the object store containing the administered object (such as initial context and location).

In the following example, the `query` subcommand is used to display information about an object whose lookup name is `myTopic`.

```
imqobjmgr query
  -l "cn=myTopic"
  -j "java.naming.factory.initial=
          com.sun.jndi.ldap.LdapCtxFactory"
  -j "java.naming.provider.url=
          ldap://mydomain.com:389/o=imq"
  -j "java.naming.security.principal=
          uid=fooUser, ou=People, o=imq"
  -j "java.naming.security.credentials=fooPasswd"
  -j "java.naming.security.authentication=simple"
```

Updating Administered Objects

You use the `update` command to modify the attributes of administered objects. You must specify the lookup name and location of the object. You use the -o option to modify attribute values.

This command changes the attributes of an administered object that represents a topic connection factory:

```
imqobjmgr update
 -t tf
 -l "cn=MyTCF"
 -o imqReconnectRetries=3
 -j "java.naming.factory.initial=
        com.sun.jndi.ldap.LdapCtxFactory"
 -j "java.naming.provider.url=
        ldap://mydomain.com:389/o=imq"
 -j "java.naming.security.principal=
        uid=fooUser, ou=People, o=imq"
 -j "java.naming.security.credentials=fooPasswd"
 -j "java.naming.security.authentication=simple"
```

Chapter 8

Security Management

This chapter explains how to perform tasks related to security, these include authentication, authorization, and encryption.

Authenticating Users You are responsible for maintaining a list of users, their groups, and passwords in a user repository. The first part of this chapter explains how you create, populate, and manage that repository. For an introduction to Sun™ ONE Message Queue (MQ) security, see "Security Manager" on page 57.

Authorizing Users You are responsible for editing a properties file that maps the user's access to broker operations to the user's name or group. The second part of this chapter explains how you can customize this properties file.

Encryption: Setting Up SSL Services Using a connection service based on the Secure Socket Layer (SSL) standard allows you to encrypt messages sent between clients and broker. For an introduction to how MQ handles encryption, see "Encryption (Enterprise Edition)" on page 59. The last part of this chapter explains how to set up an SSL-based connection service and provides additional information about using SSL.

For situations in which a password is needed for a broker to secure access to a SSL keystore, a LDAP user repository, or a JDBC-compliant persistent store, there are three means of providing such passwords:

- by having the system prompt you when the broker is started
- by passing in passwords as command line options when starting the broker (see "Starting a Broker" on page 120 and Table 5-2 on page 121)
- by storing passwords in a passfile that the system accesses when starting the broker (See "Using a Passfile" on page 204)

Authenticating Users

When a user attempts to connect to the broker, the broker authenticates the user by inspecting the name and password provided, and grants the connection if they match those in a user repository that the broker is configured to consult. This repository can be of two types:

- a flat-file repository that is shipped with MQ

  This type of user repository is very easy to use; however it is vulnerable to security attacks, and should therefore be used *only* for evaluation and development purposes. You can populate and manage the repository using the User Manager utility (`imqusermgr`). To enable authentication, you populate the user repository with each user's name, password, and the name of the user's group.

  For more information on setting up and managing the user repository, see "Using a Flat-File User Repository" on page 182.

- an LDAP server

  This could be an existing or new LDAP directory server that uses the LDAP v2 or v3 protocol for your user repository. It is not as easy to use as the flat-file repository, however it is secure, and therefore better for production environments.

  If you are using an LDAP user repository, you will need to use the tools provided by the LDAP vendor to populate and manage the user repository. For more information, see "Using an LDAP Server for a User Repository" on page 189.

Using a Flat-File User Repository

MQ provides a flat-file user repository and a command line tool, MQ User Manager (`imqusermgr`) that you can use to populate and manage the flat-file user repository. The following sections describe the flat-file user repository, its initial entries, and how you populate and manage that repository.

The default flat-file repository is located at:

`IMQ_HOME/etc/passwd` (`/etc/imq/passwd` on Solaris)

The repository is shipped with two entries (rows) already defined, as illustrated in the table below.

Table 8-1 Initial Entries in User Repository

User Name	Password	Group	State
`admin`	`admin`	`admin`	`active`
`guest`	`guest`	`anonymous`	`active`

These initial entries allow the MQ broker to be used immediately after installation without any intervention by the administrator. In other words, no initial user/password setup is required for the MQ broker to be used.

The initial `guest` user entry allows JMS clients to connect to the broker using the default `guest` user name and password (for testing purposes, for example).

The initial `admin` user entry allows you to use `imqcmd` commands to administer the broker using the default `admin` user name and password. It is recommended that you update this initial entry to change the password.

You can use the User Manager utility to edit or populate the flat-file user repository without having to first configure or start up the broker. The only requirement for using the User Manager utility is that it be run on the host where the broker is installed, and that if you want to write to the repository, you have to have the appropriate permissions:

- On Solaris, the User Manager utility can be run by the root user, or by other non-root users granted access through Solaris role based access control. To grant such access, the root user must make an entry to `/etc/user_attr` of the following form:

  username`::::type=normal;profiles=Message Queue Management`

  This adds the specified user to the MQ rights profile. To use this facility, you first must run a profile shell (`pfsh`, `pfksh`, `pfcsh`), for example,

  ```
  % /usr/bin/pfsh
  ```

  and then execute the desired User Manager (`imqusermgr`) commands.

  For more information on Solaris role based access control, see: `http://docs.sun.com/?q=Rights+Profile&p=/doc/806-4078/6jd6cjrvl&a=view`).

- On Windows, after installation, the user repository file can be written to by any user because the operating system does not control access to files using user name-based permission attributes.

The following sections explain how you populate and manage the flat-file user repository.

User Manager Utility (imqusermgr)

The User Manager utility allows you to manage a file-based user repository. This section describes the basic `imqusermgr` command syntax, provides a listing of subcommands, and summarizes `imqusermgr` command options. Subsequent sections explain how you use the `imqobjmgr` subcommands to accomplish specific tasks.

Syntax of Command

The general syntax of the `imqusermgr` command is as follows:

```
imqusermgr subcommand [options]
imqusermgr -h
imqusermgr -v
```

Note that if you specify the -v or -h options, no subcommands specified on the command line are executed. For example, if you enter the following command, version information is displayed but the `list` subcommand is not executed.

```
imqusermgr list -v
```

imqusermgr Subcommands

Table 8-2 lists the `imqusermgr` subcommands.

Table 8-2 `imqusermgr` Subcommands

Subcommand	**Description**
`add -u` *name* `-p` *passwd* `[-g` *group*`] [-s]`	Adds a user and associated password to the repository, and optionally specifies the user's group.
`delete -u` *name* `[-s] [-f]`	Deletes the specified user from the repository.
`list [-u` *name*`]`	Displays information about the specified user or all users.

`update -u` *name* `-p` *passwd* `[-a` *state*`] [-s] [-f]`
`update -u` *name* `-a` *state* `[-p` *passwd*`] [-s] [-f]`
	Updates the specified user's password and/or state.

Summary of imqusermgr Command Options

Table 8-3 lists the options to the `imqusermgr` command.

Table 8-3 `imqusermgr` Options

Option	Description
`-a` *active_state*	Specifies (`true/false`) whether the user's state should be active. A value of `true` means that the state is active. This is the default.
`-f`	Performs action without user confirmation
`-h`	Displays usage help. Nothing else on the command line is executed.
`-p` *passwd*	Specifies the user's password.
`-g` *group*	Specifies the user group. Valid values are `admin`, `user`, `anonymous`.
`-s`	Sets silent mode.
`-u` *name*	Specifies the user name.
`-v`	Displays version information. Nothing else on the command line is executed.

Groups

When adding a user entry to the repository, the administrator has the option of specifying one of three predefined groups for the user: `admin`, `user`, or `anonymous`. If no group is specified, the default group `user` is assigned.

- The *admin* group is for broker administrators. Users who are assigned this group can, by default, configure, administer, and manage the broker. The administrator can assign more than one user to the admin group.
- The *user* group is for normal (non-administrative) JMS client applications. Most MQ client applications will access the broker authenticated in the user group. As such, client applications, can produce messages to and consume messages from all topics and queues, or can browse messages in any queue by default.
- The *anonymous* group is for JMS client applications who do not wish to use a user name that is known to the broker (possibly because the application does not know of a real user name to use). This is analogous to the anonymous account present in most FTP servers. The administrator can assign only one

user to the anonymous group at any one time. It is expected that you will restrict the access privileges of this group as compared to the user group through access control or that you will remove the user from this group at deployment time.

In order to change a user's group, the administrator must delete the user entry and then add another entry for the user, specifying the new group.

You can specify access rules that define what operations the members of that group may perform. For more information, see "Authorizing Users: the Access Control Properties File" on page 192.

States

When the administrator adds a user to the repository, the user's state is active by default. To make the user inactive, the administrator must use the update command. For example, the following command makes the user JoeD inactive:

```
imqusermgr update -u JoeD -a false
```

Entries for users that have been rendered inactive are retained in the repository; however, inactive users cannot open new connections. If a user is inactive and the administrator adds another user who has the same name, the operation will fail. The administrator must delete the inactive user entry or change the new user's name or use a different name for the new user. This prevents the administrator from adding duplicate names or passwords.

Format of User Names and Passwords

User names and passwords must follow these guidelines:

- The user name and passwords may not contain the characters listed in Table 8-4.

Table 8-4 Invalid Characters for User Names and Passwords

Character	Description
*	Asterisk
,	Comma
:	Colon

- The user name and passwords may not contain a new line or carriage return as characters.
- If the name or password contains a space, the entire name or password must be enclosed in quotation marks.
- The name or password must be at least one character long.
- There is no limit on the length of passwords or user names—except for that imposed by the command shell on the maximum number of characters that can be entered on a command line.

Populating and Managing the User Repository

Use the `add` subcommand to add a user to the repository. For example, the following command adds the user, `Katharine` with the password `sesame`.

```
imqusermgr add -u Katharine -p sesame -g user
```

Use the `delete` subcommand to delete a user from the repository. For example, the following command deletes the user, `Bob`:

```
imqusermgr delete -u Bob
```

Use the `update` subcommand to change a user's password or state. For example, the following command changes Katharine's password to `alladin`:

```
imqusermgr update -u Katharine -p alladin
```

To list information about one or more users, use the `list` command. The following command shows information about the user named `isa`:

```
imqusermgr list -u isa
```

```
----------------------------------
User Name    Group     Active State
----------------------------------
isa          admin    true
```

The following command lists information about all users:

```
imqusermgr list
```

```
-------------------------------------
User Name    Group        Active State
-------------------------------------
testuser3    user         true
testuser2    user         true
testuser1    user         true
isa          admin        true
admin        admin        true
guest        anonymous    true
testuser5    user         false
testuser4    user         false
```

Changing the Default Administrator Password

For the sake of security, you must change the default password of `admin` to one that is only known to you. You need to use the `imqusermgr` tool to do this.

The following command changes the default password to `grandpoobah`.

```
imqusermgr update -u admin -p grandpoobah
```

You can quickly confirm that this change is in effect, by running any of the command line tools when the broker is running. For example, the following command should work,

```
imqcmd list svc -u admin -p grandpoobah
```

While using the old password should fail.

After changing the password, you should supply the new password when using any of the administration tools, including the administration console.

Using an LDAP Server for a User Repository

If you want to use an LDAP server for your user repository, you must set certain broker properties in the instance configuration file. These properties enable the broker to query the LDAP server for information about users and groups when a user attempts to connect to the broker or perform certain operations. The instance configuration file is located at

`IMQ_VARHOME/instances/`*brokerName*`/props/config.properties`
(`/var/imq/instances/`*brokerName*`/props/config.properties` on Solaris)

➤ To edit the configuration file to use an LDAP server

1. Specify that you are using an LDAP user repository by setting the following property:

   ```
   imq.authentication.basic.user_repository=ldap
   ```

2. Set the `imq.authentication.type` property to determine whether a password should be passed from client to broker in base64 encoding (`basic`) or in MD5 digest (`digest`). When using an LDAP directory server for a user repository, you must set the authentication type to `basic`. For example,

   ```
   imq.authentication.type=basic
   ```

3. You must also set the broker properties that control LDAP access. These properties, stored in a broker's instance configuration file, are described in Table 8-5. MQ uses JNDI API's to communicate with the LDAP directory server. Consult JNDI documentation for more information on syntax and on terms referenced in these properties. MQ 3.0.1 uses a Sun JNDI LDAP provider and uses simple authentication.

Table 8-5 LDAP-related Properties

Property	Description
`imq.user_repository.ldap.server`	The host:port for the LDAP server. Host specifies the fully qualified DNS name of the host running the directory server. Port specifies the port number that the directory server is using for communications.
`imq.user_repository.ldap.principal`	The distinguished name that the broker will use to bind to the directory server for a search. If the directory server allows anonymous searches, this property does not need to be assigned a value.

Table 8-5 LDAP-related Properties *(Continued)*

Property	Description
imq.user_repository.ldap.password	The password associated with the distinguished name used by the broker. Can only be specified in a passfile (see "Using a Passfile" on page 204). For more security, let the broker prompt you for a password, or specify the password using the following command line option: imqbrokerd -ldappassword. If the directory server allows anonymous searches, no password is needed.
imq.user_repository.ldap.base	The directory base for user entries.
imq.user_repository.ldap.uidattr	The provider-specific attribute identifier whose value uniquely identifies a user. For example: uid, cn, etc.
imq.user_repository.ldap.usrfilter	A JNDI search filter (a search query expressed as a logical expression). By specifying a search filter for users, the broker can narrow the scope of a search and thus make it more efficient. For more information, see the JNDI tutorial at the following location: http://java.sun.com/products/jndi/tutorial. This property does not have to be set.
imq.user_repository.ldap.grpsearch	A boolean specifying whether you want to enable group searches. Consult the documentation provided by your LDAP provider to determine whether you can associate users into groups. Note that nested groups are not supported in MQ 3.0.1. Default: false
imq.user_repository.ldap.grpbase	The directory base for group entries.
imq.user_repository.ldap.gidattr	The provider-specific attribute identifier whose value is a group name.
imq.user_repository.ldap.memattr	The attribute identifier in a group entry whose values are the distinguished names of the group's members.

Table 8-5 LDAP-related Properties *(Continued)*

Property	Description
`imq.user_repository.ldap.grpfiltler`	A JNDI search filter (a search query expressed as a logical expression). By specifying a search filter for groups, the broker can narrow the scope of a search and thus make it more efficient. For more information, see the JNDI tutorial at the following location. `http://java.sun.com/products/jndi/tutorial` This property does not have to be set.
`imq.user_repository.ldap.timeout`	An integer specifying (in seconds) the time limit for a search. By default this is set to 180 seconds.
`imq.user_repository.ldap.ssl.enabled`	A boolean specifying whether the broker should use the SSL protocol when talking to an LDAP server. This is set to `false` by default.

See the broker's default.properties file for a sample (default) LDAP user-repository-related properties setup.

4. If necessary, you need to edit the users/groups and rules in the access control properties file. For more information about the use of access control property files, see "Authorizing Users: the Access Control Properties File" on page 192.

5. If you want the broker to communicate with the LDAP directory server over SSL during connection authentication and group searches, you need to activate SSL in the LDAP server and then set the following properties in the broker configuration file:

   - Specify a secure port for the LDAP user repository property. For example:

     ```
     imq.user_repository.ldap.server=myhost:7878
     ```

   - Set the broker `property imq.user_repository.ldap.ssl.enabled` to `true`.

Authorizing Users: the Access Control Properties File

After connecting to the broker, the user may want to produce a message, consume a message at a destination, or browse messages at a queue destination. When the user attempts to do this, the broker checks an *access control properties file* (ACL file) to see whether the user is authorized to perform the operation. The ACL file contains rules that specify which operations a particular user (or group of users) is authorized to perform. By default, all authenticated users are allowed to produce and consume messages at any destination. You can edit the access control properties file to restrict these operations to certain users and groups.

The ACL file is used whether user information is placed in a flat-file repository or in an LDAP repository. A default ACL properties file is installed along with the broker. Its name is `accesscontrol.properties` and it is placed by the installer in the following directory:

`IMQ_HOME/etc (/etc/imq` on Solaris)

The ACL file is formatted like a Java properties file. It starts by defining the version of the file and then specifies access control rules in three sections:

- connection access control
- destination access control
- destination auto-create access control

The `version` property defines the version of the ACL properties file; you may not change this entry.

```
version=JMQFileAccessControlModel/100
```

The three sections of the ACL file that specify access control are described below, following a description of the basic syntax of access rules and an explanation of how permissions are calculated.

Access Rules Syntax

In the ACL properties file, access control defines what access specific users or groups have to protected resources like destinations and connection services. Access control is expressed by a rule or set of rules, with each rule presented as a Java property:

The basic syntax of these rules is as follows:

```
resourceType.resourceVariant.operation.access.principalType = principals
```

Table 8-6 describes the elements of syntax rules.

Table 8-6 Syntactic Elements of Access Rules

Element	Description
resourceType	One of the following: `connection`, `queue` or `topic`.
resourceVariant	An instance of the type specified by *resourceType*. For example, myQueue. The wild card character (*)may be used to mean all connection service types or all destinations.
operation	Value depends on the kind of access rule being formulated.
access	One of the following: `allow` or `deny`.
principalType	One of the following: `user` or `group`. For more information, see "Groups" on page 185.
principals	Who may have the access specified on the left-hand side of the rule. This may be an individual user or a list of users (comma delimited) if the `principalType` is `user`; it may be a single group or a list of groups (comma delimited list) if the `principalType` is `group`. The wild card character (*)may be used to represent all users or all groups.

Here are some examples of access rules:

- The following rule means that all users may send a message to the queue named q1.

  ```
  queue.q1.produce.allow.user=*
  ```

- The following rule means that any user may send messages to any queue.

  ```
  queue.*.produce.allow.user=*
  ```

NOTE To specify non-ASCII user, group, or destination names, you must use Unicode escape (`\uXXXX`) notation. If you have edited and saved the ACL file with these names in a non-ASCII encoding, you can convert the file to ASCII with the Java `native2ascii` tool. For more detailed information, see `http://java.sun.com/j2se/1.3/docs/guide/intl/faq.html`

Permission Computation

The following principles are applied when computing the permissions implied by a series of rules:

- Specific access rules override general access rules. After applying the following two rules, all can send to all queues, but Bob cannot send to tq1.

  ```
  queue.*.produce.allow.user=*
  queue.tq1.produce.deny.user=Bob
  ```

- Access given to an explicit *principal* overrides access given to a * *principal*. The following rules deny Bob the right to produce messages to tq1, but allow everyone else to do it.

  ```
  queue.tq1.produce.allow.user=*
  queue.tq1.produce.deny.user=Bob
  ```

- The * *principal* rule for users overrides the corresponding * *principal* for groups. For example, the following two rules allow all authenticated users to send messages to tq1.

  ```
  queue.tq1.produce.allow.user=*
  queue.tq1.produce.deny.group=*
  ```

- Access granted a user overrides access granted to the user's group. In the following example, if Bob is a member of User, he will be denied permission to produce messages to tq1, but all other members of User will be able to do so.

  ```
  queue.tq1.produce.allow.group=User
  queue.tq1.produce.deny.user=Bob
  ```

- Any access permission not explicitly granted through an access rule is implicitly denied. For example, if the ACL file contained no access rules, all users would be denied all operations.

- Deny and allow permissions for the same user or group cancel themselves out. For example, the following two rules result in Bob not being able to browse t1:

  ```
  queue.q1.browse.allow.user=Bob
  queue.q1.browse.deny.user=Bob
  ```

  The following two rules result in the group User not being able to consume messages at q5.

  ```
  queue.q5.consume.allow.group=User
  queue.q5.consume.deny.group=User
  ```

- When multiple same left-hand rules exist, only the last entry takes effect.

Connection Access Control

The connection access control section in the ACL properties file contains access control rules for the broker's connection services. The syntax of connection access control rules is as follows:

`connection.`*resourceVariant.access.principalType* `=` *principals*

Two values are defined for *resourceVariant*: `NORMAL` and `ADMIN`. By default all users can have access to the `NORMAL` type, but only those users whose group is `admin` may have access to `ADMIN` type connection services.

You can edit the connection access control rules to restrict a user's connection access privileges. For example, the following rules deny Bob access to `NORMAL` but allow everyone else:

```
connection.NORMAL.deny.user=Bob
connection.NORMAL.allow.user=*
```

You can use the asterisk (*) character to specify all authenticated users or groups.

You may not create your own service type; you must restrict yourself to the predefined types specified by the constants `NORMAL` and `ADMIN`.

Destination Access Control

The destination access control section of the access control properties file contains destination-based access control rules. These rules determine who (users/groups) may do what (operations) where (destinations). The types of access that are regulated by these rules include sending messages to a queue, publishing messages to a topic, receiving messages from a queue, subscribing to a topic, and browsing a messages in a queue.

By default, any user or group can have all types of access to any destination. You can add more specific destination access rules or edit the default rules. The rest of this section explains the syntax of destination access rules, which you must understand to write your own rules.

The syntax of destination rules is as follows:

resourceType.resourceVariant.operation.access.principalType = principals

Table 8-7 describes these elements:

Table 8-7 Elements of Destination Access Control Rules

Component	Description
resourceType	Must be one of `queue` or `topic`.
resourceVariant	A destination name or all destinations (*), meaning all queues or all topics.
operation	Must be one of `produce`, `consume`, or `browse`.
access	Must be one of `allow` or `deny`.
principalType	Must be one of `user` or `group`.

Access can be given to one or more users and/or one or more groups.

The following examples illustrate different kinds of destination access control rules:

- Allow all users to send messages to any queue destinations.

  ```
  queue.*.produce.allow.user=*
  ```

- Deny any member of the group `user` to subscribe to the topic Admissions.

  ```
  topic.Admissions.consume.deny.group=user
  ```

Destination Auto-Create Access Control

The final section of the ACL properties file, includes access rules that specify for which users and groups the broker will auto-create a destination.

When a user creates a producer or consumer at a destination that does not already exist, the broker will create the destination if the broker's auto-create property has been enabled and if the physical destination does not already exist.

By default, any user or group has the privilege of having a destination auto-created by the broker. This privilege is specified by the following rules:

```
queue.create.allow.user=*
```

```
topic.create.allow.user=*
```

You can edit the ACL file to restrict this type of access.

The general syntax for destination auto-create access rules is as follows:

resourceType.`create`*.access.principalType* `=` *principals*

Where *resourceType* is either `queue` or `topic`.

For example, the following rules allow the broker to auto-create topic destinations for everyone except Snoopy.

```
topic.create.allow.user=*
```

```
topic.create.deny.user=Snoopy
```

Note that the effect of destination auto-create rules must be congruent with that of destination access rules. For example, if you 1) change the destination access rule to forbid any user from sending a message to a destination but 2) enable the auto-creation of the destination, the broker *will* create the destination if it does not exist but it will *not* deliver a message to it.

Encryption: Working With an SSL Service (Enterprise Edition)

The MQ Enterprise Edition supports connection services based on the Secure Socket Layer (SSL) standard: over TCP/IP (ssljms and ssladmin) and over HTTP (httpsjms). These SSL-based connection services allow for the encryption of messages sent between clients and broker. The current MQ release supports SSL encryption based on self-signed server certificates.

To use an SSL-based connection service, you need to generate a private key/public key pair using the Key Tool utility (imqkeytool). This utility embeds the public key in a self-signed certificate that is passed to any client requesting a connection to the broker, and the client uses the certificate to set up an encrypted connection.

While MQ's SSL-based connection services are similar in concept, there are some differences in how you set them up. Secure connections over TCP/IP and over HTTP are therefore discussed separately in the following sections.

Setting Up an SSL Service Over TCP/IP

There are two SSL-based connection services that provide a direct, secure connection over TCP/IP.

ssljms This connection service is used to deliver JMS messages over a secure, encrypted connection between a client and broker.

ssladmin This connection service is used to create a secure, encrypted connection between the Command utility (`imqcmd`)—the command line administration tool—and a broker. A secure connection is not supported for the Administration Console (`imqadmin`).

➤ **To set up a ssljms connection service**

1. Generate a self-signed certificate.
2. Enable the ssljms connection service in the broker.
3. Start the broker.
4. Configure and run the client.

The procedures for setting up ssljms and ssladmin connection services are identical, except for Step 4, configuring and running the client.

Each of the steps is discussed in some detail in the sections that follow.

Step 1. Generating a Self-Signed Certificate

SSL Support in MQ 3.0.1 is oriented toward securing on-the-wire data with the assumption that the client is communicating with a known and trusted server. Therefore in MQ 3.0.1, SSL is implemented using only self-signed certificates.

Run the `imqkeytool` command to generate a self-signed certificate for the broker. The same certificate can be used for both the ssljms and ssladmin connection services. Enter the following at the command prompt:

```
imqkeytool -broker
```

The utility will prompt you for the information it needs. (On Unix systems you may need to run `imqkeytool` as the superuser (root) in order to have permission to create the keystore.)

First, `imqkeytool` prompts you for a keystore password, then it prompts you for some organizational information, and then it prompts you for confirmation. After it receives the confirmation, it pauses while it generates a key pair. It then asks you for a password to lock the particular key pair (key password); you should enter Return in response to this prompt: this makes the key password the same as the keystore password.

NOTE Remember the password you provide—you will need to provide this password later to the broker (when you start it) so it can open the keystore. You can also store the keystore password in a passfile (see “Using a Passfile” on page 204).

Running `imqkeytool` runs the JDK `keytool` utility to generate a self-signed certificate and to place it in MQ’s keystore, located at

`IMQ_HOME/etc/keystore` (`/etc/imq/keystore` on Solaris)

The keystore is in the same format as that supported by the JDK1.2 `keytool` utility.

The configurable properties for the MQ keystore are shown in Table 8-8. (For instructions on configuring these properties, see Chapter 5, “Starting and Configuring a Broker.”)

Table 8-8 Keystore Properties

Property Name	Description
`imq.keystore.file.dirpath`	For SSL-based services: specifies the path to the directory containing the keystore file. Default: `IMQ_HOME/etc` (`/etc/imq/` on Solaris)
`imq.keystore.file.name`	For SSL-based services: specifies the name of the keystore file. Default: `keystore`
`imq.keystore.password`	For SSL-based services: specifies the keystore password. Can only be stored in a passfile (see "Using a Passfile" on page 204). For more security, let the broker prompt you for the password, or specify the password using the following command line option: `imqbrokerd -password`.

You may need to regenerate a key pair in order to solve certain problems; for example:

- You forgot the keystore password.
- The SSL service fails to initialize when you start a broker and you get the exception:
  `java.security.UnrecoverableKeyException: Cannot recover key`.

  This exception may result from the fact that you had provided a key password that was different from the keystore password when you generated the self-signed certificate in "Step 1. Generating a Self-Signed Certificate" on page 199.

➤ To regenerate a key pair

1. Remove the broker's keystore, at the following location:

   `IMQ_HOME/etc/keystore` (`/etc/imq/keystore` on Solaris)

2. Rerun `imqkeytool` to generate a key pair as described in "Step 1. Generating a Self-Signed Certificate" on page 199.

Step 2. Enabling the SSL-based Service in the Broker

To enable the SSL service in the broker, you need to add `ssljms` (`ssladmin`) to the `imq.service.activelist` property.

1. Open the broker's instance configuration file. You can find it at the following location:

   `IMQ_VARHOME/instances/`*brokerName*`/props/config.properties`
   (`/var/imq/instances/`*brokerName*`/props/config.properties` on Solaris)

   where *brokerName* is the name of the broker instance.

2. Add the `ssljms` or `ssladmin` values or both (depending on the service you want) to the `imq.service.activelist` property:

   ```
   imq.service.activelist=jms,admin,httpjms,ssljms, ssladmin
   ```

Step 3. Starting the Broker

Start the broker, providing the keystore password. You can provide the password in any one of the following ways:

- Allow the broker to prompt you for the password when it starts up

  ```
  imqbrokerd
  Please enter Keystore password: mypassword
  ```

- Use the `-password` option to the `imqbrokerd` command:

  ```
  imqbrokerd -password mypassword
  ```

- Put the password in a passfile file (see "Using a Passfile" on page 204) which is accessed at broker startup. You have to first set the following broker configuration property (see "Editing the Instance Configuration File" on page 116):

  ```
  imq.passfile.enabled=true
  ```

  Once this property is set, you can access the passfile in either of two ways:

  - pass the location of the passfile to the `imqbrokerd` command:

    ```
    imqbrokerd -passfile /tmp/mypassfile
    ```

  - start the broker without the -passfile option, but specify the location of the passfile using the following two broker configuration properties:

    ```
    imq.passfile.dirpath=/tmp
    ```

    ```
    imq.passfile.name=mypassfile
    ```

For a listing of passfile-related broker properties, see Table 2-6 on page 60.

When you start a broker or client with SSL, you might notice that it consumes a lot of cpu cycles for a few seconds. This is because MQ uses JSSE (Java Secure Socket Extension) to implement SSL. JSSE uses `java.security.SecureRandom()` to generate random numbers. This method takes a significant amount of time to create the initial random number seed, and that is why you are seeing increased cpu usage. After the seed is created, the cpu level will drop to normal.

Step 4. Configuring and Running SSL-based Clients

Finally, you need to configure clients to use the secure connection services. There are two types of clients, depending on the connection service you are using: JMS clients that use ssljms, and the MQ administration Command utility (`imqcmd`) that uses ssladmin. These are treated separately in the following sections.

JMS Client

You have to make sure the client has the necessary Secure Socket Extension (JSSE) jar files in its classpath, and you need to tell it to use the `ssljms` connection service.

1. If your client is not using J2SDK1.4 (which has JSSE and JNDI support built in), make sure the client has the following jar files in its class path:

   ```
   jsse.jar, jnet.jar, jcert.jar, jndi.jar
   ```

2. Make sure the client has the following MQ jar files in its class path:

   ```
   imq.jar, jms.jar
   ```

3. Start the client and connect to the broker's ssljms service. One way to do this is by entering a command like the following:

   `java -DimqConnectionType=TLS` *clientAppName*

   Setting `imqConnectionType` tells the connection to use SSL.

   For more information on using ssljms connection services in client applications, see the chapter on using administered objects in the MQ *Developer's Guide*.

Command Utility (imqcmd)

You can establish a secure administration connection by including the `-secure` option when using `imqcmd` (see Table 6-2 on page 140) for example:

`imqcmd list svc -b` *hostName:port* `-u` *adminName* `-p` *adminPassword* `-secure`

where *adminName* and *adminPassword* are valid entries in the MQ user repository (if using a flat file repository, see "Changing the Default Administrator Password" on page 188).

Listing the connection services, as in this example, is a way to show that the `ssladmin` service is running, and that you can successfully make a secure admin connection, as shown in the following output:

```
Listing all the services on the broker specified by:

Host                 Primary Port
localhost            7676

Service Name     Port Number        Service State
admin            33984 (dynamic)    RUNNING
httpjms          -                  UNKNOWN
httpsjms         -                  UNKNOWN
jms              33983 (dynamic)    RUNNING
ssladmin         35988 (dynamic)    RUNNING
ssljms           dynamic            UNKNOWN

Successfully listed services.
```

Setting Up an SSL Service Over HTTP

In this SSL-based connection service (httpsjms), the client and broker establish a secure connection by way of a HTTPS tunnel servlet. The architecture and implementation of HTTPS support is described in Appendix B, “HTTP/HTTPS Support (Enterprise Edition)” on page 213.

Using a Passfile

In cases where you want the broker to start up without prompting you for needed passwords, or without requiring you to supply these passwords as options to the `imqbrokerd` command, you can place the needed passwords in a *passfile*.

A passfile is a simple text file containing passwords. The file is not encrypted, and therefore less secure than supplying passwords manually. Nevertheless you can set permissions on the file that limit who has access to view it. The permissions on the passfile need to give the user who starts the broker permission to read it.

A passfile can contain the passwords shown in Table 8-9:

Table 8-9 Passwords in a Passfile

Password	Description
`imq.keystore.password`	Specifies the keystore password for SSL-based services.
`imq.user_repository.ldap.password`	Specifies the password associated with the distinguished name assigned to a broker for binding to a configured LDAP user repository.
`imq.persist.jdbc.password`	Specifies the password used to open a database connection, if required.

A sample passfile can be found at the following location:

`IMQ_HOME/etc/`passfile.sample (`/etc/imq/passfile.sample` on Solaris)

Appendix A

Setting Up Plugged-in Persistence

This appendix explains how to set up a broker to use plugged-in persistence to access a JDBC-accessible data store.

Introduction

Sun™ ONE Message Queue (MQ) brokers include a Persistence Manager component that manages the writing and retrieval of persistent information (see "Persistence Manager" on page 54). The Persistence Manager is configured by default to access a built-in, file-based data store, but you can reconfigure it to plug in any data store accessible through a JDBC-compliant driver.

To configure a broker to use plugged-in persistence, you need to set a number of JDBC-related properties in the broker instance configuration file. You also need to create the appropriate database schema for performing MQ persistence operations. MQ provides a utility, Database Manager (`imqdbmgr`), which uses your JDBC driver and broker configuration properties to create and manage the plugged-in database.

The procedure described in this appendix is illustrated using, as an example, the Cloudscape DBMS bundled with the Java 2 SDK Enterprise Edition (J2EE SDK is available for download from java.sun.com). The example uses Cloudscape's embedded version (instead of the client/server version). In the procedures, instructions are illustrated using path names and property names from the Cloudscape example. They are identified by the word "Example:"

Other examples can be found at the following location:

`IMQ_HOME/demo/jdbc` (`/usr/demo/imq/jdbc` on Solaris)

Plugging In a JDBC-accessible Data Store

It takes just a few steps to plug in a JDBC-accessible data store.

➤ To plug in a JDBC-accessible data store

1. Set JDBC-related properties in the broker's configuration file.

   See the properties documented in Table A-1 on page 207.

2. Place a copy or a symbolic link to your JDBC driver jar file in the following path:

   `IMQ_VARHOME/lib/ext` (`/usr/share/lib/imq/ext/` on Solaris)

   Copy Example (Solaris):

   ```
   % cp j2sdk_install_directory/lib/cloudscape/cloudscape.jar
   IMQ_VARHOME/lib/ext
   ```

   Symbolic Link Example (Solaris):

   ```
   % ln -s j2sdk_install_directory/lib/cloudscape/cloudscape.jar
   IMQ_VARHOME/lib/ext
   ```

3. Create the database schema needed for MQ persistence.

   Use the `imqdbmgr create all` command (for an embedded database) or the `imqdbmgr create tbl` command (for an external database). See "Database Manager Utility (imqdbmgr)" on page 210.

   Example:

   `% cd IMQ_HOME/bin` (`/usr/bin` on Solaris)

   ```
   % imqdbmgr create all
   ```

NOTE If an embedded database is used, it is recommended that it be created under the following directory:

> `IMQ_VARHOME/instances/`*brokerName*`/dbstore/`*dabatabseName*.
> (/var/imq/instances/*brokerName*/`dbstore/` on Solaris)

If an embedded database is not protected by a user name and password, it is probably protected by file system permissions. To ensure that the database is readable and writable by the broker, the user who runs the broker should be the same user who created the embedded database using the imqdbmgr command (see "Database Manager Utility (imqdbmgr)" on page 210).

JDBC-related Broker Configuration Properties

The broker's instance configuration file is located in

> `IMQ_VARHOME/instances/`*brokerName*`/props/config.properties`
> (`/var/imq/instances/`*brokerName*`/props/config.properties` on Solaris)

If the file does not yet exist, you have to start up the broker using the `-name brokerName` option, for MQ to create the file.

Table A-1 presents the configuration properties that you need to set when plugging in a JDBC- accessible data store. You set these properties in the instance configuration file (`config.properties`) of each broker instance using plugged-in persistence. The table includes values you would specify for the Cloudscape DBMS example.

Table A-1 JDBC-related Properties

Property Name	Description
`imq.persist.store`	Specifies a file-based or JDBC-based data store. *Example*: `jdbc`

`imq.persist.jdbc.brokerid`
`(optional)`
	Specifies a broker instance identifier that is appended to database table names to make them unique in the case where more than one broker instance is using the same database as a persistent data store. (Usually not needed in the case of an embedded database, which stores data for only one broker instance.) The identifier must be an alphanumeric string whose length does not exceed the maximum table name length, minus 12, allowed by the database.
	Example: not needed for Cloudscape

`imq.persist.jdbc.driver`	Specifies the java class name of the JDBC driver to connect to the database. *Example*: `COM.cloudscape.core.JDBCDriver`

Table A-1 JDBC-related Properties *(Continued)*

Property Name	Description
`imq.persist.jdbc.opendburl`	Specifies the database URL for opening a connection to an existing database. *Example*: `jdbc:cloudscape:IMQ_VARHOME/instances/`*brokerName*`/dbstore/imqdb` (`jdbc:cloudscape:var/imq`... on Solaris)
`imq.persist.jdbc.createdburl (optional)`	Specifies the database URL for opening a connection to create a database. (Only specified if the database is to be created using `imqdbmgr`.) *Example*: `jdbc:cloudscape:IMQ_VARHOME/instances/`*brokerName*`/dbstore/imqdb;create=true` (`jdbc:cloudscape:var/imq`... on Solaris)

`imq.persist.jdbc.closedburl (optional)`	Specifies the database URL for shutting down the current database connection when the broker is shutdown.
	Example (required for Cloudscape):
	`jdbc:cloudscape:;shutdown=true`

`imq.persist.jdbc.user (optional)`	Specifies the user name used to open a database connection, if required. For security reasons, the value can be specified instead using command line options: `imqbrokerd -dbuser` and `imqdbmgr -u`

`imq.persist.jdbc.needpassword (optional)`	Specifies whether the database requires a password for broker access. Value of `true` means password is required. The password can be specified using the following command line options:
	`imqbrokerd -dbpassword`
	`imqdbmgr -p`
	If the password is not provided using either command line options or a passfile (see "Using a Passfile" on page 204), the broker will prompt for the password.

Table A-1 JDBC-related Properties *(Continued)*

Property Name	Description

`imq.persist.jdbc.password` `(optional)`	Specifies password used to open a database connection, if required. Can only be specified in a passfile (see "Using a Passfile" on page 204). For more security, let the broker prompt you for the password, or specify the password using the following command line options:
	`imqbrokerd -dbpassword`
	`imqdbmgr -p`

As with all broker configuration properties, values can be set using the `-D` command line option. If a database requires certain database specific properties to be set, these also can be set using the `-D` command line option when starting the broker (`imqbrokerd`) or the Database Manager utility (`imqdbmgr`).

Example:

For the Cloudscape embedded database example, instead of specifying the absolute path of a database in database connection URL's (as those shown in Table A-1 examples), the -D command line option can be used to define the Cloudscape system directory:

```
-Dcloudscape.system.home=IMQ_VARHOME/instances/brokerName/dbstore
```

In that case the URL's to create and open a database can be specified simply as:

```
imq.persist.jdbc.createdburl=jdbc:cloudscape:imqdb;create=true
```

and

```
imq.persist.jdbc.opendburl=jdbc:cloudscape:imqdb
```

respectively.

Database Manager Utility (imqdbmgr)

MQ provides a Database Manager utility (`imqdbmgr`) for setting up the schema needed for persistence. The utility can also be used to delete MQ database tables should the tables become corrupted or should you wish to use a different database as a data store.

NOTE If a broker instance crashes abnormally, unreferenced persistent information might remain in the data store. Starting up and then shutting down the broker will normally clean up the data store.

This section describes the basic `imqdbmgr` command syntax, provides a listing of subcommands, and summarizes `imqdbmgr` command options.

Syntax of Command

The general syntax of the `imqdbmgr` command is as follows:

```
imqdbmgr subcommand argument [options]
imqdbmgr -h|-help
imqdbmgr -v|-version
```

Note that if you specify the -v or -h options, no subcommands specified on the command line are executed. For example, if you enter the following command, version information is displayed but the `create` subcommand is not executed.

```
imqdbmgr create all -v
```

imqdbmgr Subcommands

The Database Manager utility (`imqdbmgr`) includes the subcommands listed in Table A-2:

Table A-2 `imqdbmgr` Subcommands

Subcommand and Argument	Description
`create all`	Creates a new database and MQ persistent storage schema. This command is used on an embedded database system, and when used, the property `imq.persist.jdbc.createdburl` needs to be specified.
`create tbl`	Creates the MQ persistent storage schema in an existing database system. This command is used on an external database system.
`delete tbl`	Deletes the existing MQ database tables in the current persistent storage database.
`recreate tbl`	Deletes the existing MQ database tables in the current persistent storage database and then re-creates the MQ persistent storage schema.

Summary of imqdbmgr Command Options

Table A-3 lists the options to the `imqdbmgr` command.

Table A-3 `imqdbmgr` Options

Option	Description
`-D`*property*`=`*value*	Sets the specified property to the specified value.
`-b` *brokerName*	Specifies the broker instance name and use the corresponding instance configuration file.
`-h`	Displays usage help. Nothing else on the command line is executed.
`-p` *password*	Specifies the database password.
`-u` *name*	Specifies the database user name.
`-v`	Displays version information. Nothing else on the command line is executed.

Appendix B

HTTP/HTTPS Support (Enterprise Edition)

The Sun™ ONE Message Queue (MQ) Enterprise Edition (see "Product Editions" on page 26) includes support for both HTTP and HTTPS connections. (HTTPS is a secure connection over HTTP, using the Secure Socket Layer standard.) This support allows client applications to communicate with the broker using the HTTP protocol instead of direct TCP connections. This appendix describes the architecture used to implement this support and explains the setup work needed to allow clients to use HTTP-based connections for MQ messaging.

HTTP/HTTPS Support Architecture

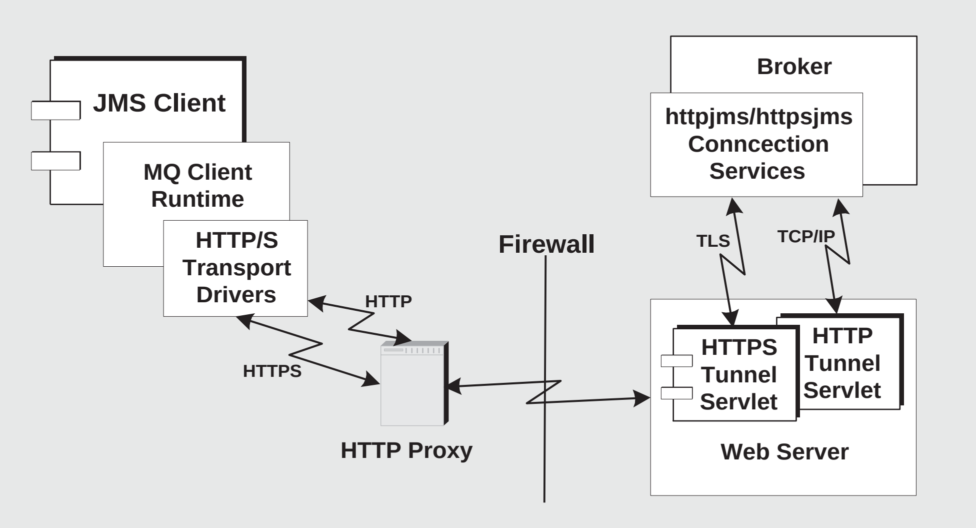
MQ messaging can be run on top of HTTP/HTTPS connections. Because HTTP/HTTPS connections are normally allowed through firewalls, this allows client applications to be separated from a broker by a firewall.

Figure B-1 on page 214 shows the main components involved in providing HTTP/HTTPS support.

- On the client side, an HTTP transport driver encapsulates the MQ message into an HTTP request and makes sure that these requests are sent to the Web server in the correct sequence.
- The JMS client can use an HTTP proxy server to communicate with the broker if necessary. The proxy's address is specified using command line options when starting the client. See "Using an HTTP Proxy" on page 219 for more information.

- An HTTP or HTTPS tunnel servlet (both bundled with MQ) is loaded in the web server and used to pull JMS messages out of client HTTP requests before forwarding them to the broker. The HTTP/HTTPS tunnel servlet also sends broker messages back to the client in response to HTTP requests made by the client. A single HTTP/HTTPS tunnel servlet can be used to access multiple brokers.

Figure B-1 HTTP/HTTPS Support Architecture



- On the broker side, the httpjms or httpsjms connection service unwraps and demultiplexes incoming messages from the corresponding tunnel servlet.
- If the Web server fails and is restarted, all connections are restored and there is no effect on clients. If the broker fails and is restarted, an exception is thrown and clients must re-establish their connections. In the unlikely case that both the Web server and the broker fail, and the broker is not restarted, the Web server will restore client connections and continue waiting for a broker connection— without notifying clients. To avoid this situation, always restart the broker.

As you can see from Figure B-1, the architecture for HTTP and HTTPS support are very similar. The main difference is that, in the case of HTTPS (httpsjms connection service), the tunnel servlet has a secure connection to both the client application and broker.

The secure connection to the broker is provided through an SSL-enabled tunnel servlet—MQ's HTTPS tunnel servlet—which passes a self-signed certificate to any broker requesting a connection. The certificate is used by the broker to set up an encrypted connection to the HTTPS tunnel servlet. Once this connection is established, a secure connection between a client application and the tunnel servlet can be negotiated by the client application and the web server.

Implementing HTTP Support

The following sections describe the steps you need to take to implement HTTP support.

➤ **To implement HTTP support**

1. Deploy the HTTP tunnel servlet on a web server.
2. Configure the broker's httpjms connection service and start the broker.
3. Configure an HTTP connection.

Step 1. Deploying the HTTP Tunnel Servlet on a Web Server

There are two general ways you can deploy the HTTP tunnel servlet on a web server:

- deploying it as a jar file—for web servers that support Servlet 2.1 or earlier
- deploying it as a web archive (WAR) file—for web servers that support Servlet 2.2 or later

Deploying as a Jar File

Deploying the MQ tunnel servlet consists of making the appropriate jar files accessible to the host web server and configuring that web server to load the servlet on startup.

The tunnel servlet jar file (`imqservlet.jar`) contains all the classes needed by the HTTP tunnel servlet and is located in the following directory:

`IMQ_HOME/lib (/usr/share/lib/imq` on Solaris)

Any web server with servlet 2.x support can be used to load this servlet. The servlet class name is:

```
com.sun.messaging.jmq.transport.
httptunnel.servlet.HttpTunnelServlet
```

The web server must be able to see the `imqservlet.jar` file. If you are planning to run the web server and the broker on different hosts, you should place a copy of the `imqservlet.jar` file in a location where the web server can access it.

You also need to configure the web server to load this servlet on startup (see "Example: Deploying the HTTP Tunnel Servlet" on page 219).

It is also recommended that you disable your web server's access logging feature in order to improve performance.

Deploying as a Web Archive File

Deploying the HTTP tunnel servlet as a WAR file consists of using the deployment mechanism provided by the web server. The HTTP tunnel servlet WAR file (`imqhttp.war`) is located in the following directory:

`IMQ_HOME/lib (/usr/share/lib/imq` on Solaris)

The WAR file includes a deployment descriptor that contains the basic configuration information needed by the web server to load and run the servlet.

Step 2. Configuring the httpjms Connection Service

HTTP support is not enabled by default for an MQ 3.0.1 broker, so you need to reconfigure the broker to enable the httpjms connection service. Once reconfigured, the broker can be started as outlined in "Starting a Broker" on page 120.

➤ To enable the httpjms connection service

1. Open the broker's instance configuration file at the following location:

   `IMQ_VARHOME/instances/`*brokerName*`/props/config.properties`
   (`/var/imq/instances/`*brokerName*`/props/config.properties` on Solaris)

   where *brokerName* is the name of the broker instance.

2. Add the `httpjms` value to the `imq.service.activelist` property:

   ```
   imq.service.activelist=jms,admin,httpjms
   ```

At startup, the broker looks for a web server and HTTP tunnel servlet running on its host machine. To access a remote tunnel servlet, however, you can reconfigure the servletHost and servletPort connection service properties.

You can also reconfigure the pullPeriod property to improve performance. The httpjms connection service configuration properties are detailed in Table B-1 on page 217.

Table B-1 httpjms Connection Service Properties

Property Name	Description
`imq.httpjms.http.servletHost`	Change this value, if necessary, to specify the name of the host (hostname or IP address) on which the HTTP tunnel servlet is running. (This can be a remote host or a specific hostname on a local host.) Default: `localhost`
`imq.httpjms.http.servletPort`	Change this value to specify the port number that the broker uses to access the HTTP tunnel servlet. (If the default port is changed on the Web server, then you must change this property accordingly.) Default: `7675`
`imq.httpjms.http.pullPeriod`	Specifies the interval, in seconds, between HTTP requests made by each client to pull messages from the broker. If the value is zero or negative, the client keeps one HTTP request pending at all times, ready to pull messages as fast as possible. With a large number of clients, this can be a heavy drain on web server resources and the server may become unresponsive. In such cases, you should set the `pullPeriod` property to a positive number of seconds. This sets the time the client's HTTP transport driver waits before making subsequent pull requests. Setting the value to a positive number conserves web server resources at the expense of the response times observed by clients. Default: `-1`

Step 3. Configuring an HTTP Connection

A client application must use an appropriately configured connection factory administered object to make an HTTP connection to a broker. This section discusses HTTP connection configuration issues.

Setting Connection Factory Attributes

To implement HTTP support, you set the following connection factory attributes (see “Connection Factory Administered Objects” on page 167):

- Set the `imqConnectionType` attribute to `HTTP`
- Set the `imqConnectionURL` to the HTTP tunnel servlet URL

  `http://`*hostName*`:`*port*`/imq/tunnel`

You can set connection factory attributes in one of the following ways:

- Using the `-o` option to the `imqobjmgr` command that creates the connection factory administered object (see “Adding a Connection Factory” on page 174), or set the attribute when creating the connection factory administered object using the Administration Console (`imqadmin`).
- Using the `-D` option to the command that launches the client application (see the MQ *Developer's Guide*).
- Using a JMS API call to set the attributes of a connection factory after you create it programmatically in client application code (see the MQ *Developer's Guide*).

Using a Single Servlet to Access Multiple Brokers

You do not need to configure multiple web servers and servlet instances if you are running multiple brokers. You can share a single web server and HTTP tunnel servlet instance among concurrently running brokers. In order to do this, you must configure the `imqConnectionURL` connection factory attribute as shown below:

`http://`*hostName:port/imq/tunnel*`?ServerName=`*hostName*`:`*brokerName*

Where *hostName* is the broker host name and *brokerName* is the name of the specific broker instance you want your client to access.

To check that you have entered the correct strings for *hostName* and *brokerName,* generate a status report for the HTTP tunnel servlet by accessing the servlet URL from a browser. The report lists all brokers being accessed by the servlet:

```
HTTP tunnel servlet ready.
Servlet Start Time : Thu May 30 01:08:18 PDT 2002
Accepting TCP connections from brokers on port : 7675
Total available brokers = 2
Broker List :
   jpgserv:broker2
   cochin:broker1
```

Using an HTTP Proxy

If you are using an HTTP proxy to access the HTTP tunnel servlet:

- Set `http.proxyHost` system property to the proxy server host name.
- Set `http.proxyPort` system property to the proxy server port number.

You can set these properties using the `-D` option to the command that launches the client application.

Example: Deploying the HTTP Tunnel Servlet

This section describes how you deploy the HTTP tunnel servlet both as a jar file and as a WAR file on the Sun ONE WEB Server. The approach you use depends on the version of Sun ONE Web Server: If it does not support Servlet 2.2 or later, it will not be able to handle WAR file deployment.

Deploying as a Jar File

The instructions below refer to deployment on Sun ONE Web Server, FastTrack Edition 4.1 using the browser-based administration GUI. This procedure consists of the following general steps:

1. add a servlet
2. configure the servlet virtual path
3. load the servlet
4. disable the servlet access log

These steps are described in the following subsections. You can verify successful HTTP tunnel servlet deployment by accessing the servlet URL using a web browser. It should display status information.

Adding a Servlet

➤ To add a tunnel servlet

1. Select the Servlets tab.
2. Choose Configure Servlet Attributes.
3. Specify a name for the tunnel servlet in the Servlet Name field.
4. Set the Servlet Code (class name) field to the following value:

   ```
   com.sun.messaging.jmq.transport.
   httptunnel.servlet.HttpTunnelServlet
   ```

5. Enter the complete path to the `imqservlet.jar` in the Servlet Classpath field. For example:

   `IMQ_HOME/lib/imqservlet.jar`
   (`/usr/share/lib/imq/imqservlet.jar` on Solaris)

6. In the Servlet args field, enter any optional arguments, as shown in Table B-2:

   Table B-2 Servlet Arguments for Deploying HTTP Tunnel Servlet Jar File

Argument	Default Value	Reference
servletHost	all hosts	See Table B-1 on page 217
servletPort	7675	See Table B-1 on page 217

   If using both arguments, separate them with a comma:

   `servletPort=`*portnumber,* `servletHost=`...

   The `serverHost` and `serverPort` argument apply only to communication between the Web Server and broker, and are set only if the default values are problematic. However, in that case, you also have to set the broker configuration properties accordingly (see Table B-1 on page 217), for example:

   ```
       imq.httpjms.http.servletPort
   ```

Configuring a Servlet Virtual Path (Servlet URL)

➤ To configure a virtual path (servlet URL) for a tunnel servlet

1. Select the Servlets tab.
2. Choose Configure Servlet Virtual Path Translation.
3. Set the Virtual Path field.

   For example, if you want the URL to be `http://`*hostName*`:`*port*`/imq/tunnel`, enter the following string in the Virtual Path field.

   ```
   /imq/tunnel
   ```

4. Set the Servlet Name field to the same value as in Step 3 in "Adding a Servlet" on page 220.

Loading a Servlet

➤ To load the tunnel servlet at web server startup

1. Select the Servlets tab.
2. Choose Configure Global Attributes.
3. In the Startup Servlets field, enter the same servlet name value as in Step 3 in "Adding a Servlet" on page 220.

Disabling a Server Access Log

You do not have to disable the server access log, but you will obtain better performance if you do.

➤ To disable the server access log

1. Select the Status tab.
2. Choose the Log Preferences Page.
3. Use the Log client accesses control to disable logging

Deploying as a WAR File

The instructions below refer to deployment on Sun ONE Web Server 6.0 Service Pack 2. You can verify successful HTTP tunnel servlet deployment by accessing the servlet URL using a web browser. It should display status information.

➤ To deploy the http tunnel servlet as a WAR file

1. In the browser-based administration GUI, select the Virtual Server Class tab and select Manage Classes.
2. Select the appropriate virtual server class name (e.g. defaultclass) and click the Manage button.
3. Select Manage Virtual Servers.
4. Select an appropriate virtual server name and click the Manage button.
5. Select the Web Applications tab.
6. Click on Deploy Web Application.
7. Select the appropriate values for the WAR File On and WAR File Path fields so as to point to the `imqhttp.war` file. It can be found in the following directory:

   `IMQ_HOME/lib` (`/usr/share/lib/imq` on Solaris)
8. Enter "`/imq`" (without the quotes) in the Application URI field.
9. Enter the installation directory path (typically somewhere under the Sun ONE Web Server installation root) where the servlet should be deployed.
10. Click `OK`.
11. Restart the web server instance.

The servlet is now available at the following address:

`http://`*hostName*`:`*port*`/imq/tunnel`

Clients can now use this URL to connect to the message service using an HTTP connection.

Implementing HTTPS Support

The following sections describe the steps you need to take to implement HTTPS support. They are similar to those in "Implementing HTTP Support" on page 215 with the addition of steps needed to generate and access SSL certificates.

➤ **To implement HTTPS support**

1. Generate a self-signed certificate for the HTTPS tunnel servlet.
2. Deploy the HTTPS tunnel servlet on a web server.
3. Configure the broker's httpsjms connection service and start the broker.
4. Configure an HTTPS connection.

Each of these steps is discussed in more detail in the sections that follow.

Step 1. Generating a Self-signed Certificate for the HTTPS Tunnel Servlet

SSL Support in MQ 3.0.1 is oriented toward securing on-the-wire data with the assumption that the client is communicating with a known and trusted server. Therefore in MQ 3.0.1, SSL is implemented using only self-signed server certificates. In the httpsjms connection service architecture, the HTTPS tunnel servlet plays the role of server to both broker and application client.

Run the `imqkeytool` utility to generate a self-signed certificate for the tunnel servlet. Enter the following at the command prompt:

```
imqkeytool -servlet keystore_location
```

The utility will prompt you for the information it needs. (On Unix systems you may need to run `imqkeytool` as the superuser (root) in order to have permission to create the keystore.)

First, `imqkeytool` prompts you for a keystore password, then it prompts you for some organizational information, and then it prompts you for confirmation. After it receives the confirmation, it pauses while it generates a key pair. It then asks you for a password to lock the particular key pair (key password); you should enter Return in response to this prompt: this makes the key password the same as the keystore password.

NOTE Remember the password you provide—you will need to provide this password later to the tunnel servlet so it can open the keystore.

Running imqkeytool runs the JDK keytool utility to generate a self-signed certificate and to place it in MQ's keystore file located as specified in the *keystore_location* argument. (The keystore is in the same keystore format as that supported by the JDK1.2 keytool.)

NOTE The HTTPS tunnel servlet must be able to see the keystore. Make sure you move/copy the generated keystore located in *keystore_location* to a location accessible by the HTTPS tunnel servlet (see "Step 2. Deploying the HTTPS Tunnel Servlet on a Web Server" on page 224).

Step 2. Deploying the HTTPS Tunnel Servlet on a Web Server

There are two general ways you can deploy the HTTPS tunnel servlet on a web server:

- deploying it as a jar file—for web servers that support Servlet 2.1 or earlier
- deploying it as a web archive (WAR) file—for web servers that support Servlet 2.2 or later

In either case, you should make sure that encryption is activated for the web server, enabling end to end secure communication between the client and broker.

Deploying as a Jar File

Deploying the MQ tunnel servlet consists of making the appropriate jar files accessible to the host web server and configuring that web server to load the servlet on startup.

The tunnel servlet jar file (imqservlet.jar) contains all the classes needed by the HTTPS tunnel servlet and is located in the following directory:

IMQ_HOME/lib (/usr/share/lib/imq on Solaris)

Any web server with servlet 2.x support can be used to load this servlet. The servlet class name is:

```
com.sun.messaging.jmq.transport.
httptunnel.servlet.HttpsTunnelServlet.
```

The web server must be able to see the imqservlet.jar file. If you are planning to run the web server and the broker on different hosts, you should place a copy of the imqservlet.jar file in a location where the web server can access it.

You also need to configure the web server to load this servlet on startup (see "Example: Deploying the HTTPS Tunnel Servlet" on page 229).

Make sure that the JSSE jar files are in the classpath for running servlets in the web server. Check the web server's documentation for how to do this.

An important aspect of configuring the web server is specifying the location and password of the self-signed certificate to be used by the HTTPS tunnel servlet to establish a secure connection with a broker. You have to place the keystore created in "Step 1. Generating a Self-signed Certificate for the HTTPS Tunnel Servlet" on page 223 in a location accessible by the HTTPS tunnel servlet.

It is also recommended that you disable your web server's access logging feature in order to improve performance.

Deploying as a Web Archive File

Deploying the HTTPS tunnel servlet as a WAR file consists of using the deployment mechanism provided by the web server. The HTTPS tunnel servlet WAR file (imqhttps.war) is located in the following directory:

IMQ_HOME/lib (/usr/share/lib/imq on Solaris)

The WAR file includes a deployment descriptor that contains the basic configuration information needed by the web server to load and run the servlet.

However, the deployment descriptor of the imqhttps.war file cannot know where you have placed the keystore file needed by the tunnel servlet (see "Step 1. Generating a Self-signed Certificate for the HTTPS Tunnel Servlet" on page 223). This requires you to edit the tunnel servlet's deployment descriptor (an XML file) to specify the keystore location before deploying the imqhttps.war file.

Step 3. Configuring the httpsjms Connection Service

HTTPS support is not enabled by default for an MQ 3.0.1 broker, so you need to reconfigure the broker to enable the httpsjms connection service. Once reconfigured, the broker can be started as outlined in "Starting a Broker" on page 120.

➤ To enable the httpsjms connection service

1. Open the broker's instance configuration file at the following location:

   `IMQ_VARHOME/instances/`*brokerName*`/props/config.properties`
   (`/var/imq/instances/`*brokerName*`/props/config.properties` on Solaris)

   where *brokerName* is the name of the broker instance.

2. Add the `httpsjms` value to the `imq.service.activelist` property:

   ```
   imq.service.activelist=jms,admin,httpsjms
   ```

At startup, the broker looks for a web server and HTTPS tunnel servlet running on its host machine. To access a remote tunnel servlet, however, you can reconfigure the servletHost and servletPort connection service properties.

You can also reconfigure the pullPeriod property to improve performance. The httpsjms connection service configuration properties are detailed in Table B-3.

Table B-3 httpsjms Connection Service Properties

Property Name	Description
`imq.httpsjms.https.servletHost`	Change this value, if necessary, to specify the name of the host (hostname or IP address) on which the HTTPS tunnel servlet is running. (This can be a remote host or a specific hostname on a local host.) Default: `localhost`
`imq.httpsjms.https.servletPort`	Change this value to specify the port number that the broker uses to access the HTTPS tunnel servlet. (If the default port is changed on the Web server, then you must change this property accordingly.) Default: `7674`
`imq.httpsjms.https.pullPeriod`	Specifies the interval, in seconds, between HTTP requests made by each client to pull messages from the broker. If the value is zero or negative, the client keeps one HTTP request pending at all times, ready to pull messages as fast as possible. With a large number of clients, this can be a heavy drain on web server resources and the server may become unresponsive. In such cases, you should set the `pullPeriod` property to a positive number of seconds. This sets the time the client's HTTP transport driver waits before making subsequent pull requests. Setting the value to a positive number conserves web server resources at the expense of the response times observed by clients. Default: `-1`

Step 4. Configuring an HTTPS Connection

A client application must use an appropriately configured connection factory administered object to make an HTTPS connection to a broker.

However, the client must also have access to SSL libraries provided by the Java Secure Socket Extension (JSSE) and must also have a root certificate. The SSL libraries are bundled with JDK 1.4. If you have an earlier JDK version, see "Configuring JSSE," otherwise proceed to "Importing a Root Certificate."

Once these issues are resolved, you can proceed to configuring the HTTPS connection.

Configuring JSSE

➤ **To configure JSSE**

1. Copy the JSSE jar files to the `JRE_HOME/lib/ext` directory.

   ```
   jsse.jar, jnet.jar, jcert.jar
   ```

2. Statically add the JSSE security provider by adding

   ```
   security.provider.n=com.sun.net.ssl.internal.ssl.Provider
   ```

   to the `JRE_HOME/lib/security/java.security` file (where *n* is the next available priority number for security provider package).

3. If not using JDK1.4, you need to set the following JSSE property using the `-D` option to the command that launches the client application:

   ```
   java.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
   ```

Importing a Root Certificate

If the root certificate of the CA who signed your web server's certificate is not in the trust database by default or if you are using a proprietary web server certificate, you have to add that certificate to the trust database. If this is the case, follow the instruction below, otherwise go to "Setting Connection Factory Attributes".

Assuming that the certificate is saved in *cert_file* and that *trust_store_file* is your keystore, run the following command:

```
JRE_HOME/bin/keytool -import -trustcacerts
-alias alias_for_certificate -file cert_file
-keystore trust_store_file
```

Answer `YES` to the question: Trust this certificate?

You also need to specify the following JSSE properties using the `-D` option to the command that launches the client application:

`javax.net.ssl.trustStore=`*trust_store_file*

`javax.net.ssl.trustStorePassword=`*trust_store_passwd*

Setting Connection Factory Attributes

To implement HTTPS support, you set the following connection factory attributes (see "Connection Factory Administered Objects" on page 167):

- Set the `imqConnectionType` attribute to `HTTP`

  The secure connection to the broker is achieved through deploying and connecting through the HTTPS tunnel servlet rather than the HTTP tunnel servlet. Both use the same connection type, however.

- Set the `imqConnectionURL` to the HTTPS tunnel servlet URL

  `https://`*hostName*`:`*port*`/imq/tunnel`

You can set connection factory attributes in one of the following ways:

- Using the `-o` option to the `imqobjmgr` command that creates the connection factory administered object (see "Adding a Connection Factory" on page 174), or set the attribute when creating the connection factory administered object using the Administration Console (`imqadmin`).
- Using the `-D` option to the command that launches the client application (see the MQ *Developer's Guide*).
- Using a JMS API call to set the attributes of a connection factory after you create it programmatically in client application code (see the MQ *Developer's Guide*).

Using a Single Servlet to Access Multiple Brokers

You do not need to configure multiple web servers and servlet instances if you are running multiple brokers. You can share a single web server and HTTPS tunnel servlet instance among concurrently running brokers. In order to do this, you must configure the `imqConnectionURL` connection factory attribute as shown below:

`https://`*hostName:port/imq/tunnel*`?ServerName=`*hostName*`:`*brokerName*

Where *hostName* is the broker host name and *brokerName* is the name of the specific broker instance you want your client to access.

To check that you have entered the correct strings for *hostName* and *brokerName,* generate a status report for the HTTPS tunnel servlet by accessing the servlet URL from a browser. The report lists all brokers being accessed by the servlet:

```
HTTPS tunnel servlet ready.
Servlet Start Time : Thu May 30 01:08:18 PDT 2002
Accepting TCP connections from brokers on port : 7674
Total available brokers = 2
Broker List :
   jpgserv:broker2
   cochin:broker1
```

Using an HTTP Proxy

If you are using an HTTP proxy to access the HTTPS tunnel servlet:

- Set `http.proxyHost` system property to the proxy server host name.
- Set `http.proxyPort` system property to the proxy server port number.

You can set these properties using the `-D` option to the command that launches the client application.

Example: Deploying the HTTPS Tunnel Servlet

This section describes how you deploy the HTTPS tunnel servlet both as a jar file and as a WAR file on the Sun ONE Web Server. The approach you use depends on the version of Sun ONE Web Server: If it does not support Servlet 2.2 or later, it will not be able to handle WAR file deployment.

Deploying as a Jar File

The instructions below refer to deployment on Sun ONE Web Server, FastTrack Edition 4.1 using the browser-based administration GUI. This procedure consists of the following general steps:

1. add a servlet
2. configure the servlet virtual path
3. load the servlet
4. disable the servlet access log

These steps are described in the following subsections. You can verify successful HTTP tunnel servlet deployment by accessing the servlet URL using a web browser. It should display status information.

Adding a Servlet

➤ To add a tunnel servlet

1. Select the Servlets tab.
2. Choose Configure Servlet Attributes.
3. Specify a name for the tunnel servlet in the Servlet Name field.
4. Set the Servlet Code (class name) field to the following value:

   ```
   com.sun.messaging.jmq.transport.
   httptunnel.servlet.HttpsTunnelServlet
   ```

5. Enter the complete path to the `imqservlet.jar` in the Servlet Classpath field. For example:

   `IMQ_HOME/lib/imqservlet.jar`
   (`/usr/share/lib/imq/imqservlet.jar` on Solaris)

6. In the Servlet args field, enter required and optional arguments, as shown in Table B-4.

Table B-4 Servlet Arguments for Deploying HTTPS Tunnel Servlet Jar File

Argument	**Default Value**	**Required?**	**See Also**
`keystoreLocation`	none	Yes	Table 8-8 on page 200
`keystorePassword`	none	Yes	Table 8-8 on page 200
`serverHost`	all hosts	No	Table B-3 on page 226
`serverPort`	`7674`	No	Table B-3 on page 226

Separate the arguments with a comma, for example:

`keystoreLocation=`*keystore_location*`,keystorePassword=`*keystore_password*`,`
`servletPort=`*portnumber*

The `serverHost` and `serverPort` argument apply only to communication between the Web Server and broker, and are set only if the default values are problematic. However, in that case, you also have to set the broker configuration properties accordingly (see Table B-3 on page 226), for example:

```
imq.httpsjms.https.servletPort
```

Configuring a Servlet Virtual Path (Servlet URL)

➤ **To configure a virtual path (servlet URL) for a tunnel servlet**

1. Select the Servlets tab.
2. Choose Configure Servlet Virtual Path Translation.
3. Set the Virtual Path field.

   For example, if you want the URL to be `http://`*hostName*`:`*port*`/imq/tunnel`, enter the following string in the Virtual Path field.

   ```
   /imq/tunnel
   ```

4. Set the Servlet Name field to the same value as in Step 3 in "Adding a Servlet" on page 230.

Loading a Servlet

➤ **To load the tunnel servlet at web server startup**

1. Select the Servlets tab.
2. Choose Configure Global Attributes.
3. In the Startup Servlets field, enter the same servlet name value as in Step 3 in "Adding a Servlet" on page 230.

Disabling a Server Access Log

You do not have to disable the server access log, but you will obtain better performance if you do.

➤ **To disable the server access log**

1. Select the Status tab.
2. Choose the Log Preferences Page.
3. Use the Log client accesses control to disable logging

Deploying as a WAR File

The instructions below refer to deployment on Sun ONE Web Server 6.0 Service Pack 2. You can verify successful HTTPS tunnel servlet deployment by accessing the servlet URL using a web browser. It should display status information.

Before deploying the HTTPS tunnel servlet, make sure that JSSE jar files are included in the web server's classpath. The simplest way to do this is to copy the `jsse.jar`, `jnet.jar`, and `jcert.jar` to `IWS60_TOPDIR/bin/https/jre/lib/ext.`

Also, before deploying the HTTPS tunnel servlet, you have to modify its deployment descriptor to point to the location where you have placed the keystore file and to specify the keystore password.

➤ **To modify the HTTPS tunnel servlet WAR file**

1. Copy the WAR file to a temporary directory.

   ```
   $ cp IMQ_HOME/lib/imqhttps.war /tmp
   ```
   (`$ cp /usr/share/lib/imq/imqhttps.war /tmp` on Solaris)

2. Make the temporary directory your current directory.

   ```
   $ cd /tmp
   ```

3. Extract the contents of the WAR file.

   ```
   $ jar xvf imqhttps.war
   ```

4. List the WAR file's deployment descriptor.

   ```
   $ ls -l WEB-INF/web.xml
   ```

5. Edit the web.xml file to provide correct values for the keystoreLocation and keystorePassword arguments (as well as serverPort and serverHost arguments, if necessary).

6. Re-assemble the contents of the WAR file.

```
$ jar uvf imqhttps.war WEB-INF/web.xml
```

You are now ready to use the modified imqhttps.war file to deploy the HTTPS tunnel servlet. (If you are concerned about exposure of the keystore password, you can use file system permissions to restrict access to the imqhttps.war file.)

➤ To deploy the https tunnel servlet as a WAR file

1. In the browser-based administration GUI, select the Virtual Server Class tab. Click Manage Classes.

2. Select the appropriate virtual server class name (e.g. defaultclass) and click the Manage button.

3. Select Manage Virtual Servers.

4. Select an appropriate virtual server name and click the Manage button.

5. Select the Web Applications tab.

6. Click on Deploy Web Application.

7. Select the appropriate values for the WAR File On and WAR File Path fields so as to point to the modified imqhttps.war file (see "To modify the HTTPS tunnel servlet WAR file" on page 232.)

8. Enter "/imq" (without the quotes) in the Application URI field.

9. Enter the installation directory path (typically somewhere under the Sun ONE Web Server installation root) where the servlet should be deployed.

10. Click OK.

11. Restart the web server instance.

The servlet is now available at the following address:

https://*hostName*:*port*/imq/tunnel

Clients can now use this URL to connect to the message service using a secure HTTPS connection.

Appendix C

Using a Broker as a Windows Service

This appendix explains how you use the Service Administrator (imqsvcadmin) utility to install, query, and remove a broker running as a Windows Service.

Running a Broker as a Windows Service

You have the option of installing a broker as a Windows service when you install Sun™ ONE Message Queue (MQ) 3.0.1. You can also use imqsvcadmin to install a broker as an Windows service after you have installed MQ 3.0.1.

Installing a broker as a Windows service means that it will start at system startup time and run in the background until you shut down. Consequently, you do not use the imqbrokerd command to start the broker—unless, you want to start an additional instance. To pass any start-up options to the broker, you can use the -args argument to the imqsvcadmin command (see Table C-2 on page 237) and specify exactly the same options you would have used for the imqbrokerd command (see "Starting a Broker" on page 120). Use the imqcmd command to control broker operations as usual.

When running as a Windows service, the Task Manager lists the broker as two executable processes. The first is imqbrokersvc.exe, which is the native Windows service wrapper. The second is the Java runtime that is actually running the broker.

Only one broker at a time can be installed and run as a Windows service.

Service Administrator Utility (imqsvcadmin)

The Service Administrator utility (`imqsvcadmin`) allows you to install, query, and remove the broker (running as a Windows service). This section describes the basic syntax of `imqsvcadmin` commands, provides a listing of subcommands, summarizes `imqsvcadmin` command options, and explains how to use these commands to accomplish specific tasks.

Syntax of Command

The general syntax of `imqsvcadmin` commands is as follows:

```
imqsvcadmin subcommand [options]

imqsvcadmin -h
```

Note that if you specify the -v, -h, or -H options, no other subcommands specified on the command line are executed. For example, if you enter the following command, help information is displayed but the `query` subcommand is not executed.

```
imqsvcadmin query -h
```

imqsvcadmin Subcommands

The MQ Service Administrator utility (`imqsvcadmin`) includes the subcommands listed in Table C-1:

Table C-1 `imqsvcadmin` Subcommands

Subcommand	Description
`install`	Installs the service and specifies startup options.
`query`	Displays the startup options to the `imqsvcadmin` command. This includes whether the service is started manually or automatically, its location, the location of the java runtime, and the value of the arguments passed to the broker on startup.
`remove`	Removes the service.

Summary of imqsvcadmin Options

Table C-2 lists the options to the `imqsvcadmin` command. For a discussion of their use, see the task-based sections that follow.

Table C-2 `imqsvcadmin` Options

Option	Description
`-h`	Displays usage help. Nothing else on the command line is executed.
`-javahome` *path*	Specifies the path to an alternate Java 2 compatible JDK. The default is to use the bundled runtime. Example: `imqsvcadmin -install -javahome d:\jdk1.4`
`-jrehome` *path*	Specifies the path to a Java 2 compatible JRE. Example: `imqsvcadmin -install -jrehome d:\jre\1.4`
`-vmargs` *arg* `[[`*arg*`]...]`	Specifies additional arguments to pass to the Java VM that is running the broker service. (You can also specify these arguments in the Windows Services Control Panel Startup Parameters field.) Example: `-vmargs "-Xms16m -Xmx128m"`

`-args` *arg* `[[`*arg*`]...]`	Specifies additional command line arguments to pass to the broker service. For a description of the `imqbrokerd` options, see "Starting a Broker" on page 120.
	(You can also specify these arguments in the Windows Services Control Panel Startup Parameters field.) For example,
	`imqsvcadmin -install`
	`        -args "-passfile d:\imqpassfile"`

The information that you specify using the `-javahome`, `-vmargs`, and `-args` options is stored in the Window's registry under the keys `JavaHome`, `JVMArgs`, and `ServiceArgs` in the path

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet
     \Services\iMQ_Broker\Parameters
```

Removing the Broker Service

Before you remove the broker service, you should use the `imqcmd shutdown bkr` command to shut down the broker. Then use the `imqsvcadmin remove` command to remove the service, and restart your computer.

Reconfiguring the Broker Service

To reconfigure the service, remove the service first, and then reinstall it, specifying different startup options with the `-args` argument.

Using an Alternate Java Runtime

You can use either the `-javahome` or `-jrehome` options to specify the location of an alternate java runtime. You can also specify these options in the Windows Services Control Panel Startup Parameters field. Note that the Startup Parameters field treats the back slash (\) as an escape character, so you will have to type it twice when using it as a path delimiter; for example, `-javahome d:\\jdk1.3`.

Querying the Broker Service

To determine the startup options for the broker service, use the `-q` option to the `imqsvcadmin` command.

```
imqsvcadmin -query

Service iMQ_Broker is installed.
Display Name: iMQ_Broker
Start Type: Manual
Binary location: c:\Program Files\Sun Microsystems\
                    Message Queue 3.0\bin\imqbrokersvc
JavaHome: c:\j2sdk1.4.0
Broker Args: -passfile d:\imqpassfile
```

Troubleshooting

If you get an error when you try and start the service, you can see error events that were logged by doing the following.

➤ **To see logged service error events**

1. Start the Event Viewer
2. Look under Log > Application.
3. Select View > Refresh to see any error events.

Location of MQ Data

Sun™ ONE Message Queue (MQ) uses many categories of data, each of which is stored in a different location, as shown in Table D-1.

Table D-1 Location of MQ 3.0 Data

Data Category	Location of Data
Broker instance configuration properties	`IMQ_VARHOME/instances/`*brokerName*`/props/config.properties` (`/var/imq/instances/`*brokerName*`/props/config.properties` on Solaris)
Persistent store (messages, destinations, durable subscriptions, transactions)	`IMQ_VARHOME/instances/`*brokerName*`/filestore/` (`/var/imq/instances/`*brokerName*`/filestore/` on Solaris) or a JDBC-accessible data store
Broker instance log files	`IMQ_VARHOME/instances/`*brokerName*`/log/` (`/var/imq/instances/`*brokerName*`/log/` on Solaris)
Administered objects (object store)	local directory of your choice or an LDAP server

Security: user repository	`IMQ_HOME/etc/passwd`
	(`/etc/imq/passwd` on Solaris)
	`or LDAP server`

Security: access control file	`IMQ_HOME/etc/accesscontrol.properties` (`/etc/imq/accesscontrol.properties` on Solaris)
Security: passfile	`IMQ_HOME/etc/`*passfileName* (`/etc/imq/`*passfileName* on Solaris)

Table D-1 Location of MQ 3.0 Data *(Continued)*

Data Category	Location of Data
Security: broker's keystore	`IMQ_HOME/etc/keystore` (`/etc/imq/keystore` on Solaris)

Appendix E

Stability of MQ Interfaces

Sun™ ONE Message Queue (MQ) uses many interfaces, that might be of use to administrators for automating administration tasks. Table E-1 classifies these interfaces according to how stable they are, that is, how unlikely they are to change in subsequent versions of the product. The classification scheme is described in Table E-2 on page 245.

Table E-1 Stability of MQ 3.0.1 Interfaces

Interface	Classification
`imqbrokerd` command line interface	Evolving
`imqadmin` command line interface	Unstable
`imqcmd` command line interface	Evolving
`imqdbmgr` command line interface	Unstable
`imqkeytool` command line interface	Evolving
`imqobjmgr` command line interface	Evolving
`imqusermgr` command line interface	Unstable
`imqobjmgr` command file	Evolving
`imqbrokerd` command	Stable
`imqadmin` command	Unstable
`imqcmd` command	Stable
`imqdbmgr` command	Unstable
`imqkeytool` command	Stable
`imqobjmgr` command	Stable
`imqusermgr` command	Unstable

Table E-1 Stability of MQ 3.0.1 Interfaces *(Continued)*

Interface	Classification
JMS API (`javax.jms`)	Standard
JAXM API (`javax.xml`)	Standard
Administered Object API (`com.sun.messaging`)	Evolving
`imq.jar` location and name	Stable
`jms.jar` location and name	Evolving
`imqbroker.jar` location and name	Private
`imqutil.jar` location and name	Private
`imqadmin.jar` location and name	Private
`imqservlet.jar` location and name	Evolving
`imqhttp.war` location and name	Evolving
`imqhttps.war` location and name	Evolving
`imqxm.jar` location and name	Evolving
`jaxm-api.jar` location and name	Evolving
`saaj-api.jar` location and name	Evolving
`saaj-impl.jar` location and name	Evolving
`activation.jar` location and name	Evolving
`mail.jar` location and name	Evolving
`dom4j.jar` location and name	Private
`fscontext.jar` location and name	Unstable
Output from `imqbrokerd`, `imqadmin`, `imqcmd`, `imqdbmgr`, `imqkeytool`, `imqobjmgr`, `imqusermgr`	Unstable
Broker log file location and content format	Unstable
passfile	Unstable
`accesscontrol.properties`	Unstable

Table E-2 Interface Stability Classification Scheme

Classification	Description
Private	Not for direct use by customers. May change or be removed in any release.
Evolving	For use by customers. Subject to incompatible change at a major (e.g. 3.0, 4.0) or minor (e.g. 3.1, 3.2) release. The changes will be made carefully and slowly. Reasonable efforts will be made to ensure that all changes are compatible but that is not guaranteed.
Stable	For use by customers. Subject to incompatible change at a major (e.g 3.0, 4.0) release only.
Standard	For use by customers. These interfaces are defined by a formal standard, and controlled by a standards organization. Incompatible changes to these interfaces are rare.
Unstable	For use by customers. Subject to incompatible change at a major (e.g. 3.0, 4.0) or minor (e.g. 3.1, 3.2) release. Customers are advised that these interfaces may be removed or changed substantially and in an incompatible way in a future release. It is recommended to customer to not create explicit dependencies on unstable interfaces.

Glossary

This glossary provides information about terms and concepts you might encounter while using Sun™ ONE Message Queue (MQ).

administered objects A pre-configured MQ object—a connection factory or a destination—created by an administrator for use by one or more JMS clients.

The use of administered objects allows JMS clients to be provider-independent; that is, it isolates them from the proprietary aspects of a provider. These objects are placed in a JNDI name space by an administrator and are accessed by JMS clients using JNDI lookups.

asynchronous communication A mode of communication in which the sender of a message need not wait for the sending method to return before it continues with other work.

authorization The process by which a message service determines whether a user can access message service resources, such as connection services or destinations.

broker The MQ entity that manages message routing, delivery, persistence, security, and logging, and which provides an interface that allows an administrator to monitor and tune performance and resource use.

JMS client An application (or software component) that interacts with other JMS clients using a message service to exchange messages.

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

client runtime See MQ client runtime.

cluster Two or more interconnected brokers that work in tandem to provide messaging services.

configuration file One or more text files containing MQ settings that are used to configure a broker. The properties are instance-specific or cluster-related.

connection 1) An active connection to an MQ message server. This can be a queue connection or a topic connection. 2) A factory for sessions that use the connection underlying MQ message server for producing and consuming messages.

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

consume The receipt of a message taken from a destination by a message consumer.

consumer An object (MessageConsumer) created by a session that is used for receiving messages 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.

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.

delivery policy A specification of how a queue is to route messages when more than one message consumer is registered. The policies are: single, failover, and round-robin.

destination The physical destination in an MQ message server to which produced messages are delivered for routing and subsequent delivery to consumers. This physical destination is identified and encapsulated by an administered object that a JMS 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.

MQ client runtime Software that provides JMS clients with an interface to the MQ message server. The client runtime supports all operations needed for clients to send messages to destinations and to receive messages from such destinations.

MQ message server Software that provides delivery services for an MQ messaging system, including connections to JMS clients, message routing and delivery, persistence, security, and logging. The message server maintains physical destinations to which JMS clients send messages, and from which the messages are delivered to consuming clients.

JMS (Java Message Service) A standard set of interfaces and semantics that define how a JMS client accesses the facilities of a message service. These interfaces provide a standard way for Java programs to create, send, receive, and read messages.

JMS provider A product that implements the JMS interfaces for a messaging system and adds the administrative and control functions needed for a complete product.

message selector A way for a consumer to select messages based on property values (selectors) in JMS message headers. A message service performs message filtering and routing based on criteria placed in message selectors.

message service See MQ message server.

messages Asynchronous requests, reports, or events that are consumed by JMS 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.

point-to-point delivery model Producers address messages to specific queues; consumers extract messages from queues established to hold their messages. A message is delivered to only one message consumer.

produce Passing a message to the client runtime for delivery to a destination.

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).

publish/subscribe delivery model Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to a topic. The system distributes messages arriving from a topic's multiple publishers to its multiple subscribers.

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.

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 which must either be completed or entirely rolled back.

user group The group to which the user of a JMS client belongs for purposes of authorizing access to MQ message server resources, such as connections and destinations.

Index