Open Message Queue

Administration Guide

Release 5.0

May 2013

This guide provides background and information needed by system administrators to set up and manage an Open Message Queue 5.0 messaging system.

Open Message Queue Administration Guide, Release 5.0

Contents

Part II Administrative Tasks

3 Starting Brokers and Clients

4 Configuring a Broker

5 Managing a Broker

6 Configuring and Managing Connection Services

7 Managing Message Delivery

8 Configuring Persistence Services

9 Configuring and Managing Security Services

10 Configuring and Managing Broker Clusters

11 Managing Administered Objects

12 Configuring and Managing Bridge Services

13 Monitoring Broker Operations

14 Analyzing and Tuning a Message Service

15 Troubleshooting

Part III   Reference

16   Command Line Reference

17   Broker Properties Reference

18   Physical Destination Property Reference

19   Administered Object Attribute Reference

20 JMS Resource Adapter Property Reference

21 Metrics Information Reference

22 JES Monitoring Framework Reference

Part IV Appendixes

A Distribution-Specific Locations of Message Queue Data

B Stability of Message Queue Interfaces

C HTTP/HTTPS Support

D   JMX Support

E   Frequently Used Command Utility Commands

List of Examples

List of Figures

List of Tables

Preface

This *Administration Guide* provides background and information needed by system administrators to set up and manage an Open Message Queue messaging system.

This preface consists of the following sections:

- Who Should Use This Book
- Before You Read This Book
- How This Book Is Organized
- Documentation Conventions
- Related Documentation
- Documentation, Support, and Training
- Documentation Accessibility

Who Should Use This Book

This guide is intended for administrators and application developers who need to perform Message Queue administrative tasks. A Message Queue administrator is responsible for setting up and managing a Message Queue messaging system, especially the message broker at the heart of the system.

Before You Read This Book

Before reading this guide, you should read the *Open Message Queue Technical Overview* to become familiar with the Message Queue implementation of the Java Message Service specification, with the components of the Message Queue service, and with the basic process of developing, deploying, and administering a Message Queue application.

How This Book Is Organized

The following table describes the contents of this manual.

Chapter/Appendix	Description
Introduction to Message Queue Administration	
Chapter 1, "Administrative Tasks and Tools"	Introduces Message Queue administrative tasks and tools.

Chapter/Appendix	Description
Chapter 2, "Quick-Start Tutorial"	Provides a hands-on tutorial to acquaint you with the Message Queue Administration Console.
Administrative Tasks	
Chapter 3, "Starting Brokers and Clients"	Describes how to start the Message Queue broker and clients.
Chapter 4, "Configuring a Broker"	Describes how configuration properties are set and read, and gives an introduction to the configurable aspects of the broker.
Chapter 5, "Managing a Broker"	Describes broker management tasks.
Chapter 6, "Configuring and Managing Connection Services"	Describes configuration and management tasks relating to the broker's connection services.
Chapter 7, "Managing Message Delivery"	Describes how to create and manage physical destinations and manage other aspects of message delivery.
Chapter 8, "Configuring Persistence Services"	Describes how to set up a file-based or JDBC-based data store to perform persistence services.
Chapter 9, "Configuring and Managing Security Services"	Describes security-related tasks, such as managing password files, authentication, authorization, and encryption.
Chapter 10, "Configuring and Managing Broker Clusters"	Describes how to set up and manage a cluster of Message Queue brokers.
Chapter 11, "Managing Administered Objects"	Describes the object store and shows how to perform tasks related to administered objects (connection factories and destinations).
Chapter 12, "Configuring and Managing Bridge Services"	Describes how to set up and manage The JMS and STOMP bridge services.
Chapter 13, "Monitoring Broker Operations"	Describes how to set up and use Message Queue monitoring facilities.
Chapter 14, "Analyzing and Tuning a Message Service"	Describes techniques for analyzing and optimizing message service performance.
Chapter 15, "Troubleshooting"	Provides suggestions for determining the cause of common Message Queue problems and the actions you can take to resolve them.
Reference	
Chapter 16, "Command Line Reference"	Provides syntax and descriptions for Message Queue command line utilities.
Chapter 17, "Broker Properties Reference"	Describes the configuration properties of Message Queue message brokers.
Chapter 18, "Physical Destination Property Reference"	Describes the configuration properties of physical destinations.
Chapter 19, "Administered Object Attribute Reference"	Describes the configuration properties of administered objects (connection factories and destinations).
Chapter 20, "JMS Resource Adapter Property Reference"	Describes the configuration properties of the Message Queue Resource Adapter for use with an application server.
Chapter 21, "Metrics Information Reference"	Describes the metric information that a Message Queue message broker can provide for monitoring, turning, and diagnostic purposes. .
Chapter 22, "JES Monitoring Framework Reference"	Lists Message Queue attributes that are accessible by means of the Java Enterprise System Monitoring Framework (JESMF).
Appendixes	

Chapter/Appendix	Description
Appendix A, "Distribution-Specific Locations of Message Queue Data"	Lists the locations of Message Queue files and provides information about the location of Message Queue files in previous releases.
Appendix B, "Stability of Message Queue Interfaces"	Describes the stability of various Message Queue interfaces.
Appendix C, "HTTP/HTTPS Support"	Describes how to set up and use the Hypertext Transfer Protocol (HTTP) for Message Queue communication.
Appendix D, "JMX Support"	Describes Message Queue's administrative support for client programs using the Java Management Extensions (JMX) application programming interface
Appendix E, "Frequently Used Command Utility Commands"	Lists some frequently used Message Queue Command utility (`imqcmd`) commands.

Documentation Conventions

This section describes the following conventions used in Message Queue documentation:

- Typographic Conventions
- Symbol Conventions
- Shell Prompt Conventions
- Directory Variable Conventions

Typographic Conventions

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

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

Symbol Conventions

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

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

Shell Prompt Conventions

The following table shows the conventions used in Message Queue documentation for the default UNIX system prompt and superuser prompt for the C shell, Bourne shell, Korn shell, and for the Windows operating system.

Shell	Prompt
C shell on UNIX, Linux, or AIX	*machine-name*%
C shell superuser on UNIX, Linux, or AIX	*machine-name*#
Bourne shell and Korn shell on UNIX, Linux, or AIX	$
Bourne shell and Korn shell superuser on UNIX, Linux, or AIX	#
Windows command line	`C:\>`

Directory Variable Conventions

Message Queue documentation makes use of three directory variables; two of which represent environment variables needed by Message Queue. (How you set the environment variables varies from platform to platform.)

The following table describes the directory variables that might be found in this book and how they are used. Some of these variables refer to the directory *mqInstallHome*, which is the directory where Message Queue is installed to when using the installer or unzipped to when using a zip-based distribution.

> **Note:** In this book, directory variables are shown without platform-specific environment variable notation or syntax (such as `$IMQ_HOME` on UNIX). Non-platform-specific path names use UNIX directory separator (/) notation.

Variable	Description
IMQ_HOME	The Message Queue home directory:
	■ For installations of Message Queue bundled with GlassFish Server, IMQ_HOME is *as-install-parent*/mq, where *as-install-parent* is the parent directory of the GlassFish Server base installation directory, glassfish3 by default.
	■ For installations of Open Message Queue, IMQ_HOME is *mqInstallHome*/mq.
IMQ_VARHOME	The directory in which Message Queue temporary or dynamically created configuration and data files are stored; IMQ_VARHOME can be explicitly set as an environment variable to point to any directory or will default as described below:
	■ For installations of Message Queue bundled with GlassFish Server, IMQ_VARHOME defaults to *as-install-parent*/glassfish/domains/domain1/imq.
	■ For installations of Open Message Queue, IMQ_HOME defaults to *mqInstallHome*/var/mq.
IMQ_JAVAHOME	An environment variable that points to the location of the Java runtime environment (JRE) required by Message Queue executable files. By default, Message Queue looks for and uses the latest JDK, but you can optionally set the value of IMQ_JAVAHOME to wherever the preferred JRE resides.

Related Documentation

The information resources listed in this section provide further information about Message Queue in addition to that contained in this manual. The section covers the following resources:

- Message Queue Documentation Set
- Java Message Service (JMS) Specification
- JavaDoc
- Example Client Applications
- Online Help

Message Queue Documentation Set

The documents that constitute the Message Queue documentation set are listed in the following table in the order in which you might normally use them. These documents are available through the Oracle GlassFish Server documentation web site at

- `http://www.oracle.com/technetwork/indexes/documentation/index.html`

Document	Audience	Description
Technical Overview	Developers and administrators	Describes Message Queue concepts, features, and components.
Release Notes	Developers and administrators	Includes descriptions of new features, limitations, and known bugs, as well as technical notes.
Administration Guide	Administrators, also recommended for developers	Provides background and information needed to perform administration tasks using Message Queue administration tools.

Document	Audience	Description
Developer's Guide for Java Clients	Developers	Provides a quick-start tutorial and programming information for developers of Java client programs using the Message Queue implementation of the JMS or SOAP/JAXM APIs.
Developer's Guide for C Clients	Developers	Provides programming and reference documentation for developers of C client programs using the Message Queue C implementation of the JMS API (C-API).
Developer's Guide for JMX Clients	Administrators	Provides programming and reference documentation for developers of JMX client programs using the Message Queue JMX API.

Java Message Service (JMS) Specification

The Message Queue message service conforms to the Java Message Service (JMS) application programming interface, described in the *Java Message Service Specification.* This document can be found at the URL

- http://www.oracle.com/technetwork/java/jms/index.html

JavaDoc

JMS and Message Queue API documentation in JavaDoc format is included in Message Queue installations at `IMQ_HOME/javadoc/index.html`. This documentation can be viewed in any HTML browser. It includes standard JMS API documentation as well as Message Queue-specific APIs.

Example Client Applications

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

Example Java Client Applications

Example Java client applications are included in Message Queue installations at `IMQ_HOME/examples`. See the `README` files located in this directory and its subdirectories for descriptive information about the example applications.

Example C Client Programs

Example C client applications are included in Message Queue installations at `IMQ_HOME/examples/C`. See the `README` files located in this directory and its subdirectories for descriptive information about the example applications.

Example JMX Client Programs

Example Java Management Extensions (JMX) client applications are included in Message Queue installations at `IMQ_HOME/examples/jmx`. See the `README` files located in this directory and its subdirectories for descriptive information about the example applications.

Online Help

Online help is available for the Message Queue command line utilities; for details, see Command Line Reference for details. The Message Queue graphical user interface (GUI) administration tool, the Administration Console, also includes a context-sensitive help facility; see Administration Console Online Help.

Documentation, Support, and Training

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

- Documentation (http://docs.oracle.com)

- Support (http://www.oracle.com/us/support/044752.html)

- Training (http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=315)

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Part I

Introduction to Message Queue Administration

- Chapter 1, "Administrative Tasks and Tools"
- Chapter 2, "Quick-Start Tutorial"

1

Administrative Tasks and Tools

This chapter provides an overview of Open Message Queue administrative tasks and the tools for performing them, focusing on common features of the command line administration utilities. It consists of the following sections:

- Administrative Tasks
- Administration Tools

Administrative Tasks

The typical administrative tasks to be performed depend on the nature of the environment in which you are running Message Queue. The demands of a software development environment in which Message Queue applications are being developed and tested are different from those of a production environment in which such applications are deployed to accomplish useful work. The following sections summarize the typical administrative requirements of these two different types of environment.

Administration in a Development Environment

In a development environment, the emphasis is on flexibility. The Message Queue message service is needed principally for testing applications under development. Administration is generally minimal, with programmers often administering their own systems. Such environments are typically distinguished by the following characteristics:

- Simple startup of brokers for use in testing
- Administered objects instantiated in client code rather than created administratively
- Auto-created destinations
- File-system object store
- File-based persistence
- File-based user repository
- No master broker in multiple-broker clusters

Administration in a Production Environment

In a production environment in which applications must be reliably deployed and run, administration is more important. Administrative tasks to be performed depend on the complexity of the messaging system and of the applications it must support. Such

tasks can be classified into two general categories: setup operations and maintenance operations.

Setup Operations

Administrative setup operations in a production environment typically include some or all of the following:

Administrator security

- Setting the password for the default administrative user (admin) (Changing a User's Password)

- Controlling individual or group access to the administrative connection service (Authorization Rules for Connection Services) and the dead message queue (Authorization Rules for Physical Destinations)

- Regulating administrative group access to a file-based or Lightweight Directory Access Protocol (LDAP) user repository (User Groups and Status, Using an LDAP User Repository)

General security

- Managing the contents of a file-based user repository (Using the User Manager Utility) or configuring the broker to use an existing LDAP user repository (Using an LDAP User Repository)

- Controlling the operations that individual users or groups are authorized to perform (User Authorization)

- Setting up encryption services using the Secure Socket Layer (SSL) (Message Encryption)

Administered objects

- Setting up and configuring an LDAP object store (LDAP Server Object Stores)

- Creating connection factories and destinations (Adding Administered Objects)

Broker clusters

- Creating a cluster configuration file (The Cluster Configuration File)

- Designating a master broker (Managing a Conventional Cluster's Configuration Change Record)

Persistence

- Configuring a broker to use a persistent store (Configuring Persistence Services).

Memory management

- Setting a destination's configuration properties to optimize its memory usage (Updating Physical Destination Properties, Physical Destination Property Reference)

Maintenance Operations

Because application performance, reliability, and security are at a premium in production environments, message service resources must be tightly monitored and controlled through ongoing administrative maintenance operations, including the following:

Broker administration and tuning

- Using broker metrics to tune and reconfigure a broker (Analyzing and Tuning a Message Service)

- Managing broker memory resources (Managing Broker System-Wide Memory)

- Creating and managing broker clusters to balance message load (Configuring and Managing Broker Clusters)

- Recovering failed brokers (Starting Brokers).

Administered objects

- Adjusting connection factory attributes to ensure the correct behavior of client applications (Connection Factory Attributes)

- Monitoring and managing physical destinations (Configuring and Managing Physical Destinations)

- Controlling user access to destinations (Authorization Rules for Physical Destinations)

Client management

- Monitoring and managing durable subscriptions (see Managing Durable Subscriptions).

- Monitoring and managing transactions (see Managing Transactions).

Administration Tools

This section describes the tools you use to configure and manageMessage Queue broker services. The tools fall into two categories:

- Built-in Administration Tools

- JMX-Based Administration

Built-in Administration Tools

Message Queue's built-in administration tools include both command line and GUI tools:

- Command Line Utilities

- Administration Console

Command Line Utilities

All Message Queue utilities are accessible via a command line interface. Utility commands share common formats, syntax conventions, and options. These utilities allow you to perform various administrative tasks, as noted below, and therefore can require different administrative permissions:

- The **Broker utility** (`imqbrokerd`) starts up brokers and specifies their configuration properties, including connecting them together into a cluster. Permissions: User account that initially started the broker.

- The **Command utility** (`imqcmd`) controls brokers and their resources and manages physical destinations. Permissions: Message Queue admin user account.

- The **Object Manager utility** (`imqobjmgr`) manages provider-independent *administered objects* in an object store accessible via the Java Naming and Directory Interface (JNDI). Permissions: Object store (flat-file or LDAP server) access account.

- The **Database Manager utility** (`imqdbmgr`) creates and manages databases for persistent storage that conform to the Java Database Connectivity (JDBC) standard. Permissions: JDBC database manager account.

- The **User Manager utility** (`imqusermgr`) populates a file-based user repository for user authentication and authorization. Permissions: user account that initially started the broker.

- The **Service Administrator utility** (`imqsvcadmin`) installs and manages a broker as a Windows service. Permissions: Administrator.

- The **Key Tool utility** (`imqkeytool`) generates self-signed certificates for Secure Socket Layer (SSL) authentication. Permissions: root user (Solaris and Linux platforms) or Administrator (Windows platform).

The executable files for the above command line utilities are in the `IMQ_HOME/bin` directory.

See Command Line Reference for detailed information on the use of these utilities.
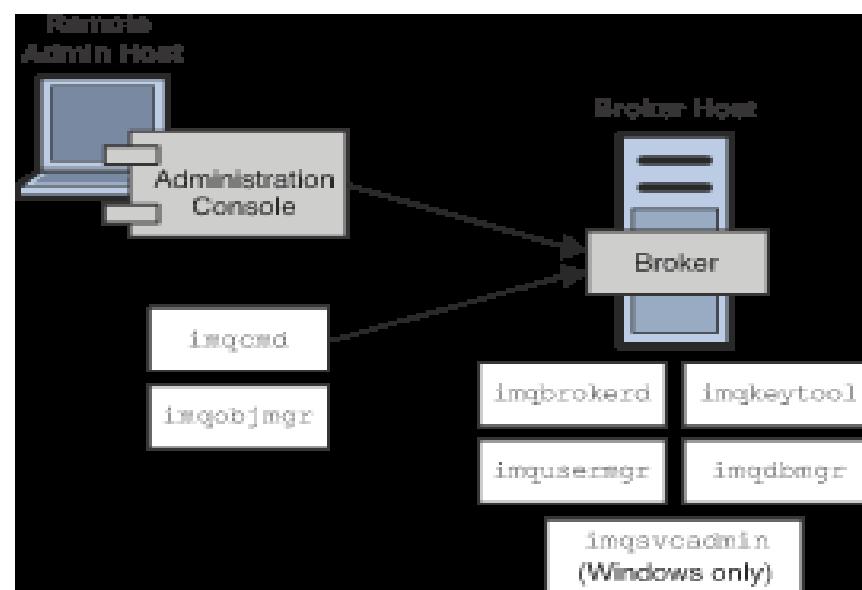
Administration Console

The Message Queue Administration Console combines some of the capabilities of the Command and Object Manager utilities. You can use it to perform the following tasks:

- Connect to and control a broker remotely

- Create and manage physical destinations

- Create and manage administered objects in a JNDI object store

However, you cannot use the Administration Console to perform such tasks as starting up a broker, creating broker clusters, managing a JDBC database or a user repository, installing a broker as a Windows service, or generating SSL certificates. For these, you need the other command line utilities (Broker, Database Manager, User Manager, Service Administrator, and Key Tool), which cannot operate remotely and must be run on the same host as the broker they manage (see Figure 1–1).

Figure 1–1   Local and Remote Administration Utilities

See Quick-Start Tutorial for a brief, hands-on introduction to the Administration Console. More detailed information on its use is available through its own help facility.

JMX-Based Administration

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

- *Resources* include everything that you can manipulate using the Command utility (`imqcmd`) and the Message Queue Admin Console: the broker, connection services, connections, destinations, durable subscribers, transactions, and so on.

- *Management* includes the ability to dynamically configure and monitor resources, and the ability to obtain notifications about state changes and error conditions.

JMX is the Java standard for building management applications. Message Queue is based on the JMX 1.2 specification, which is part of JDK 1.5.

For information on the broker's JMX infrastructure and how to configure the broker to support JMX client applications, see JMX Support.

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

2

Quick-Start Tutorial

This quick-start tutorial provides a brief introduction to Message Queue administration by guiding you through some basic administrative tasks using the Message Queue Administration Console, a graphical interface for administering a message broker and object store. The chapter consists of the following sections:

- Starting the Administration Console
- Administration Console Online Help
- Working With Brokers
- Working With Physical Destinations
- Working With Object Stores
- Working With Administered Objects
- Running the Sample Application

The tutorial sets up the physical destinations and administered objects needed to run a simple JMS-compliant application, `HelloWorldMessageJNDI`. The application is available in the `helloworld` subdirectory of the example applications directory, `IMQ_HOME/examples`. In the last part of the tutorial, you will run this application.

> **Note:** You must have the Message Queue product installed in order to follow the tutorial.

The tutorial is only a basic introduction; it is not a substitute for reading the documentation. By following the steps described in the tutorial, you will learn how to

- Start a Message Queue broker
- Connect to a broker and use the Administration Console to manage it
- Create physical destinations on the broker
- Create an object store and use the Administration Console to connect to it
- Add administered objects to the object store and view their properties

> **Note:** The instructions given in this tutorial are specific to the Windows platform. Where necessary, supplemental notes are added for users of other platforms.

Some administrative tasks cannot be accomplished using the Administration Console. You must use command line utilities to perform such tasks as the following:

- Start up a broker

- Create a broker cluster

- Configure certain physical destination properties

- Manage a JDBC database for persistent storage

- Manage a user repository

- Install a broker as a Windows service

- Generate SSL certificates

All of these tasks are covered in later chapters of this manual.
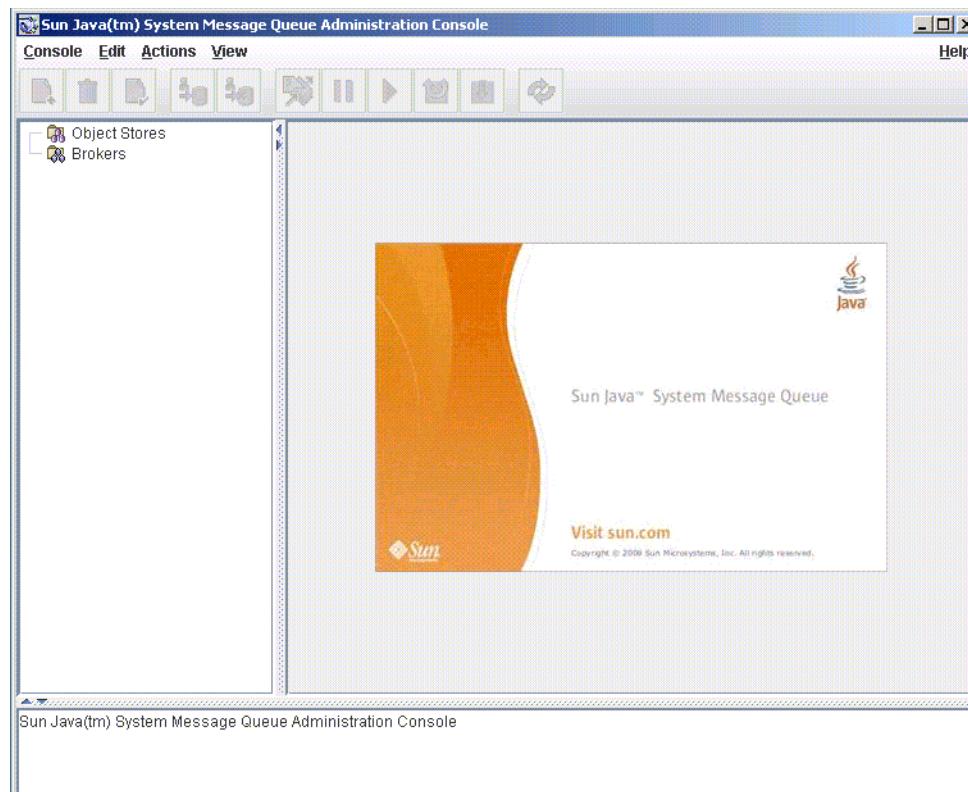
Starting the Administration Console

To start the Administration Console, enter the command:

```
IMQ_HOME/bin/imqadmin
```
You may need to wait a few seconds before the Administration Console window is displayed (see Figure 2–1).

Figure 2–1   Administration Console Window
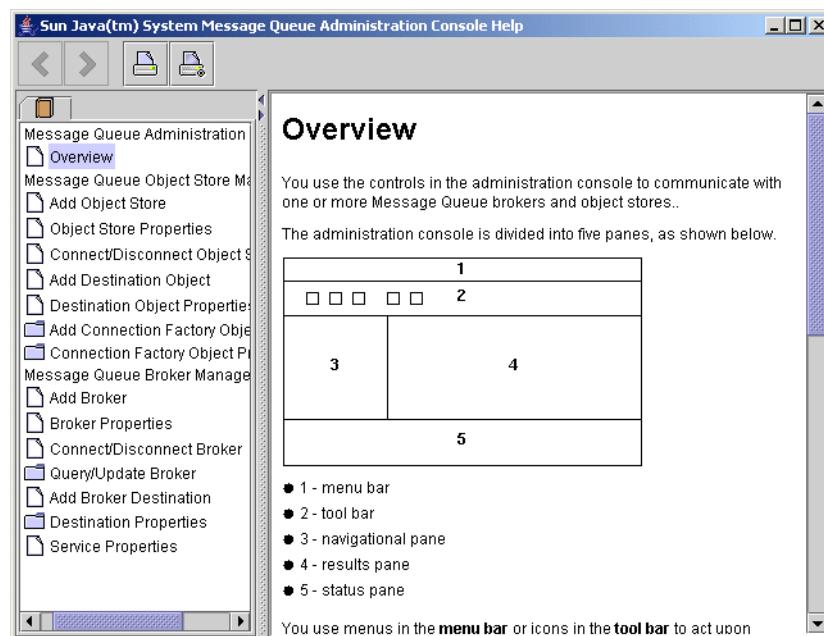
Take a few seconds to examine the Administration Console window. It has a menu bar at the top, a tool bar just below it, a navigation pane to the left, a result pane to the right (now displaying graphics identifying the Message Queue product), and a status pane at the bottom.

> **Note:** As you work with the Administration Console, you can use the Refresh command on the View menu to update the visual display of any element or group of elements, such as a list of brokers or object stores.

Administration Console Online Help

The Administration Console provides a help facility containing complete information about how to use the Console to perform administrative tasks. To use the help facility, pull down the Help menu at the right end of the menu bar and choose Overview. The Administration Console's Help window (Figure 2–2) will be displayed.

Figure 2–2   Administration Console Help Window

The Help window's navigation pane, on the left, organizes topics into three areas: Message Queue Administration Console, Message Queue Object Store Management, and Message Queue Broker Management. Within each area are files and folders. The folders provide help for dialog boxes containing multiple tabs, the files for simple dialog boxes or individual tabs. When you select an item in the navigation pane, the result pane to the right shows the contents of that item. With the Overview item chosen, the result pane displays a skeletal view of the Administration Console window identifying each of the window's panes, as shown in the figure.

Your first task with the Administration Console will be to create a reference to a broker. Before you start, however, check the Help window for information. Click the Add Broker item in the Help window's navigation pane; the contents of the result pane will change to show text explaining what it means to add a broker and describing the use of each field in the Add Broker dialog box. Read through the help text, then close the Help window.

Working With Brokers

This section describes how to use the Administration Console to connect to and manage message brokers.

Starting a Broker

You cannot start a broker using the Administration Console. Instead, enter the command:

```
IMQ_HOME/bin/imqbrokerd
```
If you used the Windows Start menu, the command window will appear, indicating that the broker is ready by displaying lines like the following:

```
Loading persistent data...    Broker "imqbroker@stan:7676 ready.
```
Reactivate the Administration Console window. You are now ready to add the broker to the Console and connect to it. You do not have to start the broker before adding a reference to it in the Administration Console, but you must start it before you can connect to it.

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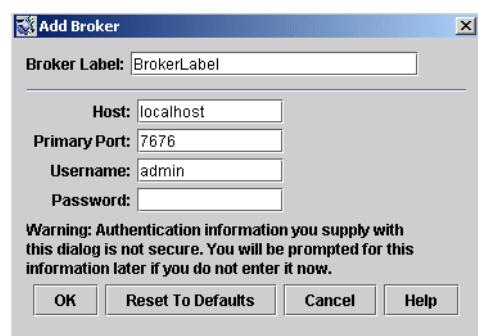
Adding a broker creates a reference to that broker in the Administration Console. After adding the broker, you can connect to it.

To Add a Broker to the Administration Console

1. Click on the Brokers item in the Administration Console window's navigation pane and choose Add Broker from the Actions menu.

   Alternatively, you can right-click on Brokers and choose Add Broker from the pop-up context menu. In either case, the Add Broker dialog box (Figure 2–3) will appear.

Figure 2–3   Add Broker Dialog Box

2. Enter a name for the broker in the Broker Label field.

   This provides a label that identifies the broker in the Administration Console.
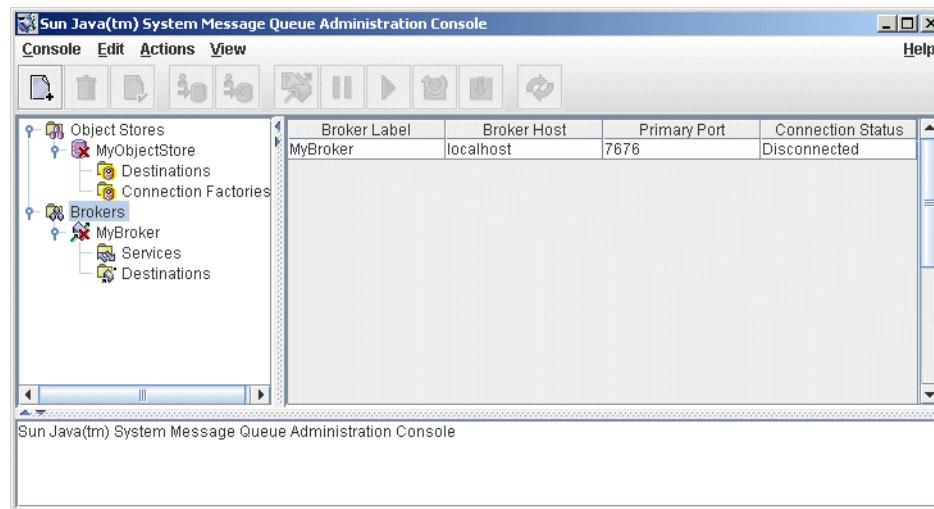
   Note the default host name (`localhost`) and primary port (`7676`) specified in the dialog box. These are the values you must specify later, when you configure the connection factory that the client will use to create connections to this broker.

   For this exercise, type the name `MyBroker` into the Broker Label field. 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 and dismiss the dialog box.

The new broker will appear under Brokers in the navigation pane, as shown in Figure 2–4. The red X over the broker's icon indicates that it is not currently connected to the Administration Console.

Figure 2–4   Broker Displayed in Administration Console Window

Once you have added a broker, you can use the Properties command on the Actions menu (or the pop-up context menu) to display a Broker Properties dialog box, similar to the Add Broker dialog shown in Adding a Broker to the Administration Console, to view or modify any of its properties.

Connecting to a Broker

Now that you have added a broker to the Administration Console, you can proceed to connect to it.

To Connect to a Broker

1. Click on the broker's name in the Administration Console window's navigation pane and choose Connect to Broker from the Actions menu.

Alternatively, you can right-click on the broker's name and choose Connect to Broker from the pop-up context menu. In either case, the Connect to Broker dialog box (Figure 2–5) will appear.

Figure 2–5   Connect to Broker Dialog Box

2. Enter the user name and password with which to connect to the broker.

The dialog box initially displays the default user name, admin . In a real-world environment, you should establish secure user names and passwords as soon as possible (see User Authentication); for this exercise, simply use the default value.

The password associated with the default user name is also `admin`; type it into the Password field in the dialog box. This will connect you to the broker with administrative privileges.

3. Click OK to connect to the broker and dismiss the dialog box.

Once you have connected to the broker, you can use the commands on the Actions menu (or the context menu) to perform the following operations on a selected broker:

- Pause Broker temporarily suspends the operation of a running broker.

- Resume Broker resumes the operation of a paused broker.

- Restart Broker reinitializes and restarts a broker.

- Shut Down Broker terminates the operation of a broker.

- Query/Update Broker displays or modifies a broker's configuration properties.

- Disconnect from Broker terminates the connection between a broker and the Administration Console.

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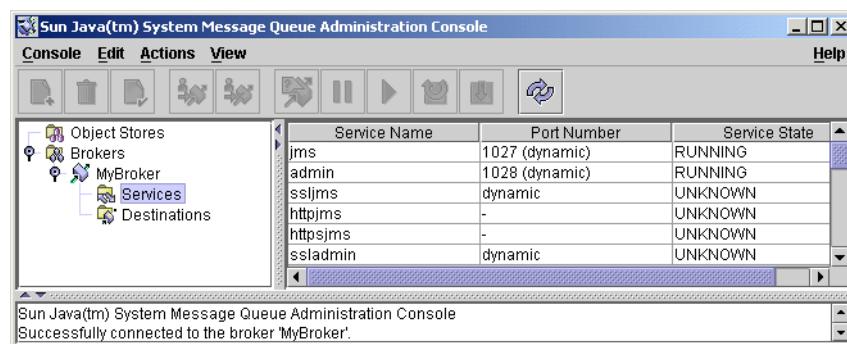
A broker is distinguished by the connection services it provides and the physical destinations it supports.

To View Available Connection Services

1. Select Services under the broker's name in the Administration Console window's navigation pane.

   A list of the available services will appear in the result pane (see Figure 2–6), showing the name, port number, and current state of each service.

   Figure 2–6   Viewing Connection Services
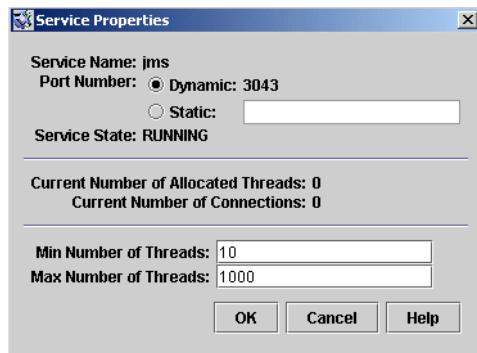
2. Select a service by clicking on its name in the result pane.

   For this exercise, select the name `jms`.

3. Choose Properties from the Actions menu.

   The Service Properties dialog box (Figure 2–7) will appear. You can use this dialog box to assign the service a static port number and to change the minimum and maximum number of threads allocated for it.

Figure 2–7   Service Properties Dialog Box

For this exercise, do not change any of the connection service's properties.

4. Click OK to accept the new property values and dismiss the dialog box.

The Actions menu also contains commands for pausing and resuming a service. If you select the admin service and pull down the Actions menu, however, you will see that the Pause Service command is disabled. This is because the admin service is the Administration Console's link to the broker: if you paused it, you would no longer be able to access the broker.

Working With Physical Destinations

A *physical destination* is a location on a message broker where messages received from a message producer are held for later delivery to one or more message consumers. Destinations are of two kinds, depending on the *messaging domain* in use: *queues* (point-to-point domain) and *topics* (publish/subscribe domain). See the *Open Message Queue Technical Overview* for further discussion of messaging domains and the destinations associated with them.
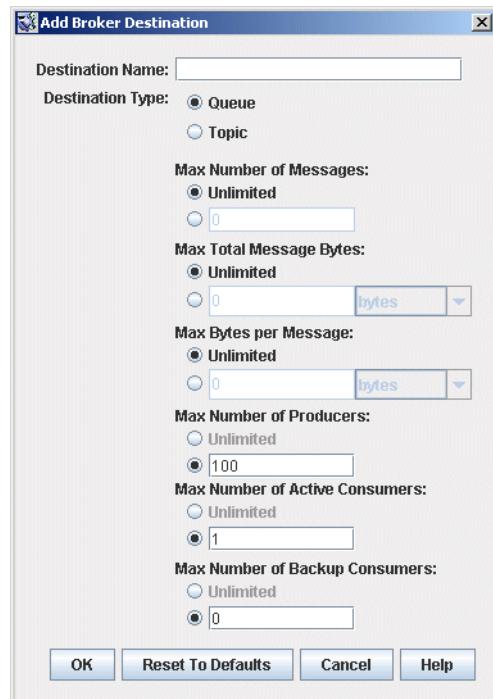
Creating a Physical Destination

By default, message brokers are configured to create new physical destinations automatically whenever a message producer or consumer attempts to access a nonexistent destination. Such *auto-created destinations* are convenient to use while testing client code in a software development environment. In a production setting, however, it is advisable to disable the automatic creation of destinations and instead require all destinations to be created explicitly by an administrator. The following procedure shows how to add such an *admin-created destination* to a broker.

To Add a Physical Destination to a Broker

1. Click on the Destinations item under the broker's name in the Administration Console window's navigation pane and choose Add Broker Destination from the Actions menu.

Alternatively, you can right-click on Destinations and choose Add Broker Destination from the pop-up context menu. In either case, the Add Broker Destination dialog box (Figure 2–8) will appear.

Figure 2–8   Add Broker Destination Dialog Box

2.   Enter a name for the physical destination in the Destination Name field.

Note the name that you assign to the destination; you will need it later when you create an administered object corresponding to this physical destination.

For this exercise, type in the name `MyQueueDest`.

3.   Select the Queue or Topic radio button to specify the type of destination to create.

For this exercise, select Queue if it is not already selected.

4.   Click OK to add the physical destination and dismiss the dialog box.

The new destination will appear in the result pane.

Viewing Physical Destination Properties

You can use the Properties command on the Administration Console's Actions menu to view or modify the properties of a physical destination.

To View or Modify the Properties of a Physical Destination

1.   Select Destinations under the broker's name in the Administration Console window's navigation pane.
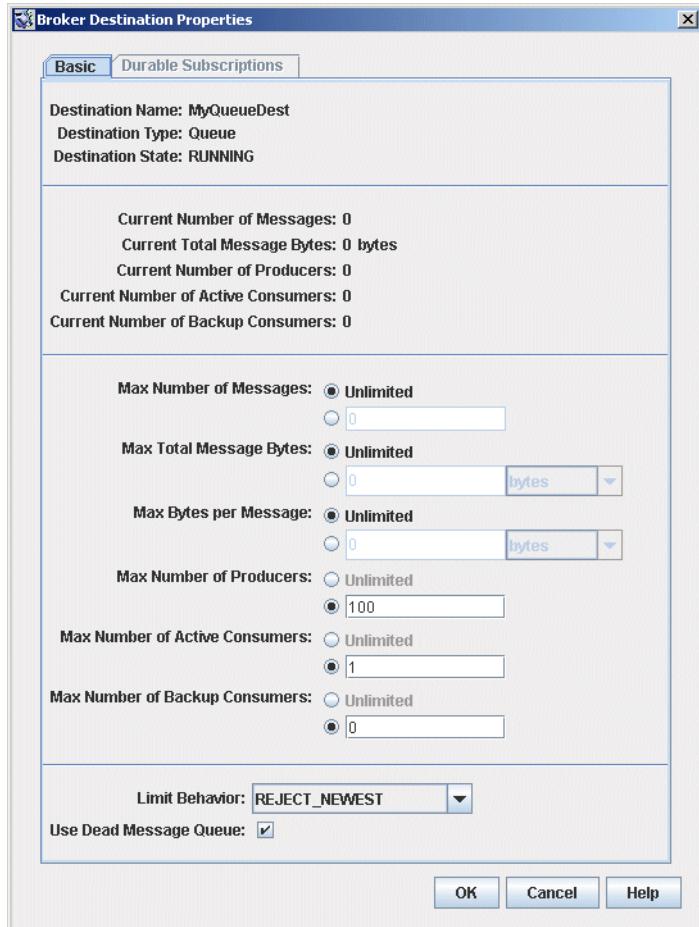
A list of the available physical destinations will appear in the result pane, showing the name, type, and current state of each destination.

2.   Select a physical destination by clicking on its name in the result pane.

3.   Choose Properties from the Actions menu.

The Broker Destination Properties dialog box (Figure 2–9) will appear, showing current status and configuration information about the selected physical destination. You can use this dialog box to change various configuration

properties, such as the maximum number of messages, producers, and consumers that the destination can accommodate.

Figure 2–9   Broker Destination Properties Dialog Box

For this exercise, do not change any of the destination's properties.

For topic destinations, the Broker Destination Properties dialog box contains an additional tab, Durable Subscriptions. Clicking on this tab displays information for Durable Subscriptions, such as the Subscription Name, Client ID, and Number of Messages.

You can use the Durable Subscriptions panel's Purge and Delete buttons to

- Purge all pending messages associated with a durable subscription
- Remove a durable subscription from the topic

The Durable Subscriptions tab is disabled for queue destinations.

4. Click OK to accept the new property values and dismiss the dialog box.

Purging Messages From a Physical Destination

Purging messages from a physical destination removes all pending messages associated with the destination, leaving the destination empty.

To Purge Messages From a Physical Destination

1. Select Destinations under the broker's name in the Administration Console window's navigation pane.

   A list of the available physical destinations will appear in the result pane, showing the name, type, and current state of each destination.

2. Select a destination by clicking on its name in the result pane.

3. Choose Purge Messages from the Actions menu.

   A confirmation dialog box will appear, asking you to confirm that you wish to proceed with the operation.

4. Click Yes to confirm the operation and dismiss the confirmation dialog.

Deleting a Physical Destination

Deleting a destination purges all of its messages and then destroys the destination itself, removing it permanently from the broker to which it belongs.

To Delete a Physical Destination

1. Select Destinations under the broker's name in the Administration Console window's navigation pane.

   A list of the available destinations will appear in the result pane, showing the name, type, and current state of each destination.

2. Select a destination by clicking on its name in the result pane.

3. Choose Delete from the Edit menu.

   A confirmation dialog box will appear, asking you to confirm that you wish to proceed with the operation.

4. Click Yes to confirm the operation and dismiss the confirmation dialog.

   For this exercise, do not delete the destination `MyQueueDest` that you created earlier; instead, click No to dismiss the confirmation dialog without performing the delete operation.

Working With Object Stores

An *object store* is used to store Message Queue *administered objects,* which encapsulate implementation and configuration information specific to a particular Message Queue provider. An object store can be either a Lightweight Directory Access Protocol (LDAP) directory server or a directory in the local file system.

Although it is possible to instantiate and configure administered objects directly from within a client application's code, it is generally preferable to have an administrator create and configure these objects and store them in an object store, where client applications can access them using the Java Naming and Directory Interface (JNDI). This allows the client code itself to remain provider-independent.

Adding an Object Store

Although the Administration Console allows you to *manage* an object store, you cannot use it to *create* one; the LDAP server or file-system directory that will serve as the object store must already exist ahead of time. You can then add this existing object
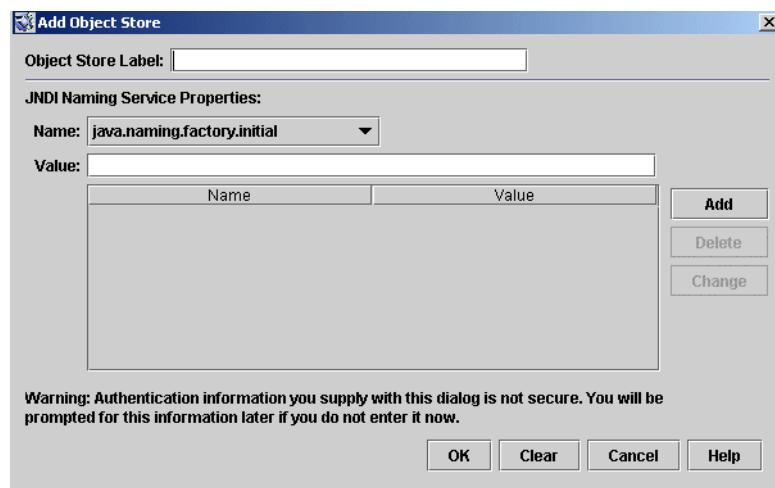
store to the Administration Console, creating a reference to it that you can use to operate on it from within the Console.

> **Note:** The sample application used in this chapter assumes that the object store is held in a directory named `Temp` on the C drive. If you do not already have a folder named `Temp` on your C drive, create one before proceeding with the following exercise. (On non-Windows platforms, you can use the `/tmp` directory, which should already exist.)

To Add an Object Store to the Administration Console

1. Click on the Object Stores item in the Administration Console window's navigation pane and choose Add Object Store from the Actions menu.

   Alternatively, you can right-click on Object Stores and choose Add Object Store from the pop-up context menu. In either case, the Add Object Store dialog box (Figure 2–10) will appear.

Figure 2–10   Add Object Store Dialog Box
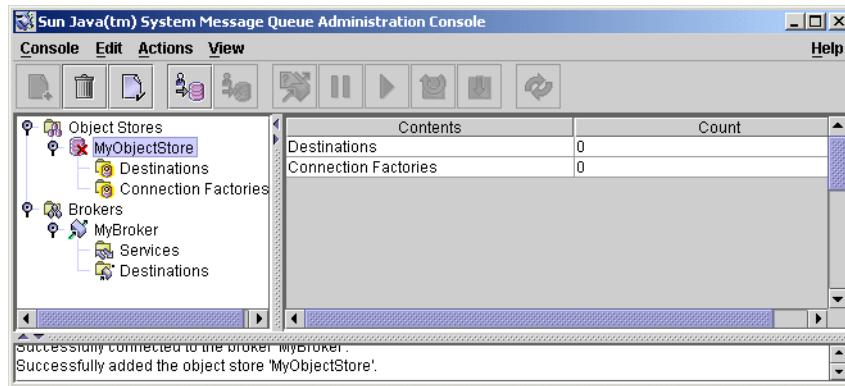
2. Enter a name for the object store in the Object Store Label field.

   This provides a label that identifies the object store in the Administration Console.

   For this exercise, type in the name `MyObjectStore`.

3. Enter the JNDI attribute values to be used for looking up administered objects:

   a. Select the name of the attribute you wish to specify from the Name pull-down menu.

   b. Type the value of the attribute into the Value field.

   c. Click the Add button to add the specified attribute value.

   The property and its value will appear in the property summary pane.

   Repeat steps Adding an Object Store to Adding an Object Store for as many attributes as you need to set.

   For this exercise, set the `java.naming.factory.initial` attribute to

   `com.sun.jndi.fscontext.RefFSContextFactory`

and the `java.naming.provider.url` attribute to

`file:///C:/Temp`
(or `file:///tmp` on the Solaris or Linux platforms). These are the only attributes you need to set for a file-system object store; see LDAP Server Object Stores for information on the attribute values needed for an LDAP store.

4. Click OK to add the object store and dismiss the dialog box.

The new object store will appear under Object Stores in the navigation pane, as shown in Figure 2–11. The red X over the object store's icon indicates that it is not currently connected to the Administration Console.

Figure 2–11   Object Store Displayed in Administration Console Window

When you click on the object store in the navigation pane, its contents are listed in the result pane. Since you have not yet added any administered objects to the object store, the Count column shows 0 for both destinations and connection factories.

Once you have added an object store, you can use the Properties command on the Actions menu (or the pop-up context menu) to display an Object Store Properties dialog box, similar to the Add Object Store dialog shown in Figure 2–10, to view or modify any of its properties.

Connecting to an Object Store

Now that you have added an object store to the Administration Console, you must connect to it in order to add administered objects to it.

To Connect to an Object Store

1. Click on the object store's name in the Administration Console window's navigation pane and choose Connect to Object Store from the Actions menu.

Alternatively, you can right-click on the object store's name and choose Connect to Object Store from the pop-up context menu. In either case, the red X will disappear from the object store's icon, indicating that it is now connected to the Administration Console.

Working With Administered Objects

Once you have connected an object store to the Administration Console, you can proceed to add administered objects (connection factories and destinations) to it. This section describes how.

> **Note:** The Administration Console displays only Message Queue administered objects. If an object store contains a non-Message Queue object with the same lookup name as an administered object that you want to add, you will receive an error when you attempt the add operation.

Adding a Connection Factory
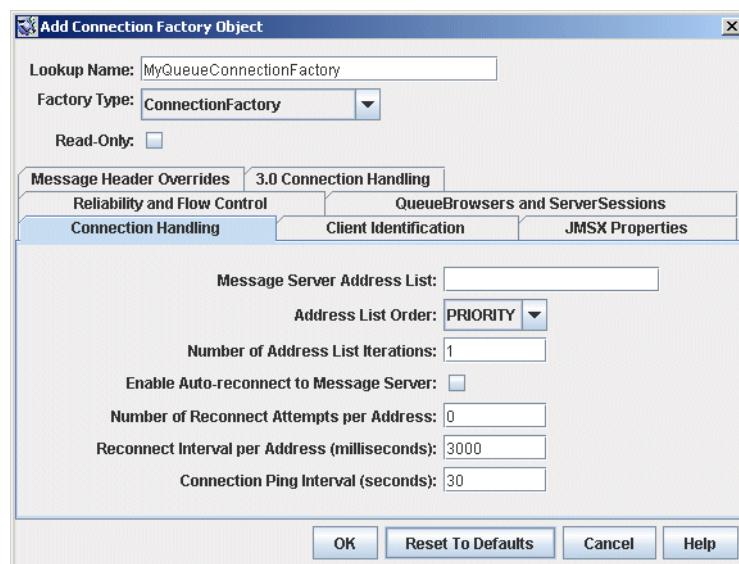
Connection factories are used by client applications to create connections to a broker. By configuring a connection factory, you can control the properties of the connections it creates.

To Add a Connection Factory to an Object Store

1. Make sure the object store is connected to the Administration Console (see Connecting to an Object Store).

2. Click on the Connection Factories item under the object store's name in the Administration Console window's navigation pane and choose Add Connection Factory Object from the Actions menu.

   Alternatively, you can right-click on Connection Factories and choose Add Connection Factory Object from the pop-up context menu. In either case, the Add Connection Factory Object dialog box (Figure 2–12) will appear.

Figure 2–12   Add Connection Factory Object Dialog Box

3. Enter a name for the connection factory in the Lookup Name field.

   This is the name that client applications will use when looking up the connection factory with JNDI.

   For this exercise, type in the name `MyQueueConnectionFactory`.

4. Choose the type of connection factory you wish to create from the Factory Type pull-down menu.

   For this exercise, choose QueueConnectionFactory.

5. Click the Connection Handling tab.

   The Connection Handling panel will appear, as shown in Figure 2–12.

6. Fill in the Message Server Address List field with the address(es) of the broker(s) to which this connection factory will create connections.

   The address list may consist of a single broker or (in the case of a broker cluster) multiple brokers. For each broker, it specifies information such as the broker's connection service, host name, and port number. The exact nature and syntax of the information to be specified varies, depending on the connection service to be used; see Connection Handling for specifics.

   For this exercise, there is no need to type anything into the Message Server Address List field, since the sample application HelloWorldMessageJNDI expects the connection factory to use the standard address list attributes to which it is automatically configured by default (connection service jms , host name localhost, and port number 7676).

7. Configure any other attributes of the connection factory as needed.

   The Add Connection Factory Object dialog box contains a number of other panels besides Connection Handling, which can be used to configure various attributes for a connection factory.

   For this exercise, do not change any of the other attribute settings. You may find it instructive, however, to click through the other tabs to get an idea of the kinds of configuration information that can be specified. Use the Help button to learn more about the contents of these other configuration panels.

8. If appropriate, click the Read-Only checkbox.

   This locks the connection factory object's configuration attributes to the values they were given at creation time. A read-only administered object's attributes cannot be overridden, whether programmatically from client code or administratively from the command line.

   For this exercise, do not check Read-Only.

9. Click OK to create the connection factory, add it to the object store, and dismiss the dialog box.

   The new connection factory will appear in the result pane.

Adding a Destination

A *destination* administered object represents a physical destination on a broker, enabling clients to send messages to that physical destination independently of provider-specific configurations and naming syntax. When a client sends a message addressed via the administered object, the broker will deliver the message to the corresponding physical destination, if it exists. If no such physical destination exists, the broker will create one automatically if auto-creation is enabled, as described under Creating a Physical Destination, and deliver the message to it; otherwise, it will generate an error signaling that the message cannot be delivered.

The following procedure describes how to add a destination administered object to the object store corresponding to an existing physical destination.

To Add a Destination to an Object Store

1. Make sure the object store is connected to the Administration Console (see Connecting to an Object Store).

2. Click on the Destinations item under the object store's name in the Administration Console window's navigation pane and choose Add Destination Object from the Actions menu.

   Alternatively, you can right-click on Destinations and choose Add Destination Object from the pop-up context menu. In either case, the Add Destination Object dialog box (Figure 2–13) will appear.

Figure 2–13   Add Destination Object Dialog Box

3. Enter a name for the destination administered object in the Lookup Name field.

   This is the name that client applications will use when looking up the destination with JNDI.

   For this exercise, type in the name `MyQueue`.

4. Select the Queue or Topic radio button to specify the type of destination object to create.

   For this exercise, select Queue if it is not already selected.

5. Enter the name of the corresponding physical destination in the Destination Name field.

   This is the name you specified when you added the physical destination to the broker (see Working With Physical Destinations).

   For this exercise, type in the name `MyQueueDest`.

6. Optionally, enter a brief description of the destination in the Destination Description field.

   The contents of this field are intended strictly for human consumption and have no effect on client operations.

   For this exercise, you can either delete the contents of the Destination Description field or type in some descriptive text such as

   `Example destination for MQ Admin Guide tutorial`

7. If appropriate, click the Read-Only checkbox.
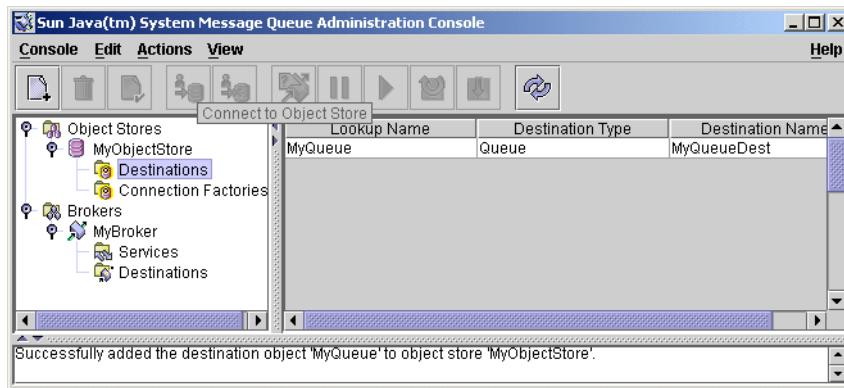
   This locks the destination object's configuration attributes to the values they were given at creation time. A read-only administered object's attributes cannot be overridden, whether programmatically from client code or administratively from the command line.

For this exercise, do not check Read-Only.

8. Click OK to create the destination object, add it to the object store, and dismiss the dialog box.

The new destination object will appear in the result pane, as shown in Figure 2–14.

Figure 2–14   Destination Object Displayed in Administration Console Window

Viewing Administered Object Properties

You can use the Properties command on the Administration Console's Actions menu to view or modify the properties of an administered object.

To View or Modify the Properties of an Administered Object

1. Select Connection Factories or Destinations under the object store's name in the Administration Console window's navigation pane.

A list of the available connection factory or destination administered objects will appear in the result pane, showing the lookup name and type of each (as well as the destination name in the case of destination administered objects).

2. Select an administered object by clicking on its name in the result pane.

3. Choose Properties from the Actions menu.

The Connection Factory Object Properties or Destination Object Properties dialog box will appear, similar to the Add Connection Factory Object (Figure 2–12) or Add Destination Object (Figure 2–13) dialog. You can use this dialog box to change the selected object's configuration attributes. Note, however, that you cannot change the object's lookup name; the only way to do this is the delete the object and then add a new administered object with the desired lookup name.

4. Click OK to accept the new attribute values and dismiss the dialog box.

Deleting an Administered Object

Deleting an administered object removes it permanently from the object store to which it belongs.

To Delete an Administered Object

1. Select Connection Factories or Destinations under the object store's name in the Administration Console window's navigation pane.

A list of the available connection factory or destination administered objects will appear in the result pane, showing the lookup name and type of each (as well as the destination name in the case of destination administered objects).

2. Select an administered object by clicking on its name in the result pane.

3. Choose Delete from the Edit menu.

A confirmation dialog box will appear, asking you to confirm that you wish to proceed with the operation.

4. Click Yes to confirm the operation and dismiss the confirmation dialog.

For this exercise, do not delete the administered objects `MyQueue` or `MyQueueConnectionFactory` that you created earlier; instead, click No to dismiss the confirmation dialog without performing the delete operation.

Running the Sample Application

The sample application `HelloWorldMessageJNDI` is provided for use with this tutorial. It uses the physical destination and administered objects that you created:

- A queue physical destination named `MyQueueDest`

- A queue connection factory administered object with JNDI lookup name `MyQueueConnectionFactory`

- A queue administered object with JNDI lookup name `MyQueue`

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

Before running the application, open the source file `HelloWorldMessageJNDI.java` and read through the code. The program is short and amply documented; you should have little trouble understanding how it works.

To Run the Sample Application

1. Make the directory containing the `HelloWorldmessageJNDI` application your current directory:

   ```
   cd IMQ_HOME/examples/helloworld/helloworldmessagejndi
   ```
   You should find the file `HelloWorldMessageJNDI.class` present. (If you make changes to the application, you must recompile it using the procedure for compiling a client application given in the *Open Message Queue Developer's Guide for Java Clients*.)

2. Set the `CLASSPATH` variable to include the current directory containing the file `HelloWorldMessageJNDI.class`, as well as the following `.jar` files that are included in the Message Queue product:

   - `jms.jar`

   - `imq.jar`

   - `jndi.jar`

   - `fscontext.jar`

   See the *Open Message Queue Developer's Guide for Java Clients* for information on setting the `CLASSPATH` variable.

> **Note:** The file `jndi.jar` is bundled with JDK 1.4. You need not add this file to your `CLASSPATH` unless you are using an earlier version of the JDK.

3. Run the `HelloWorldMessageJNDI` application by executing one of the following commands (depending on the platform you're using):

   ■ On Solaris or Linux:

   ```
   % java HelloWorldMessageJNDI file:///tmp
   ```
   ■ On Windows:

   ```
   java HelloWorldMessageJNDI
   ```
   If the application runs successfully, you should see the output shown in Example 2–1.

Example 2–1   Output from Sample Application

```
java HelloWorldMessageJNDI
Using file:///C:/Temp for Context.PROVIDER_URL


Looking up Queue Connection Factory object with lookup name:
MyQueueConnectionFactory
Queue Connection Factory object found.
Looking up Queue object with lookup name: MyQueue
Queue object found.


Creating connection to broker.
Connection to broker created.

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

Part II

Administrative Tasks

3

Starting Brokers and Clients

After installing Message Queue and performing some preparatory steps, you can begin starting brokers and clients. A broker's configuration is governed by a set of configuration files, which can be overridden by command line options passed to the Broker utility (`imqbrokerd`); see Configuring a Broker for more information.

This chapter contains the following sections:

- Preparing System Resources
- Starting Brokers
- Deleting a Broker Instance
- Starting Clients

Preparing System Resources

Before starting a broker, there are two preliminary system-level tasks to perform: synchronizing system clocks and (on the Solaris or Linux platform) setting the file descriptor limit. The following sections describe these tasks.

Synchronizing System Clocks

Before starting any brokers or clients, it is important to synchronize the clocks on all hosts that will interact with the Message Queue system. Synchronization is particularly crucial if you are using message expiration (time-to-live). Time stamps from clocks that are not synchronized could prevent message expiration from working as expected and prevent the delivery of messages. Synchronization is also crucial for broker clusters.

Configure your systems to run a time synchronization protocol, such as Simple Network Time Protocol (SNTP). Time synchronization is generally supported by the `xntpd` daemon on Solaris and Linux, and by the `W32Time` service on Windows. (See your operating system documentation for information about configuring this service.) After the broker is running, avoid setting the system clock backward.

Setting the File Descriptor Limit

On the Solaris and Linux platforms, the shell in which a client or broker is running places a soft limit on the number of file descriptors that a process can use. In Message Queue, each connection a client makes, or a broker accepts, uses one of these file descriptors. Each physical destination that has persistent messages also uses a file descriptor.

As a result, the file descriptor limit constrains the number of connections a broker or client can have. By default, the maximum is 256 connections on Solaris or 1024 on Linux. (In practice, the connection limit is actually lower than this because of the use of file descriptors for persistent data storage.) If you need more connections than this, you must raise the file descriptor limit in each shell in which a client or broker will be executing. For information on how to do this, see the man page for the ulimit command.

Starting Brokers

You can start a broker either interactively, using the Message Queue command line utilities or the Windows Start menu, or by arranging for it to start automatically at system startup. The following sections describe how.

Starting Brokers Interactively

You can start a broker interactively from the command line, using the Broker utility (imqbrokerd). (Alternatively, on Windows, you can start a broker from the Start menu.) You *cannot* use the Administration Console (imqadmin) or the Command utility (imqcmd) to start a broker; the broker must already be running before you can use these tools.

On the Solaris and Linux platforms, a broker instance must always be started by the same user who initially started it. Each broker instance has its own set of configuration properties and file-based persistent data store. When the broker instance first starts, Message Queue uses the user's file creation mode mask (umask) to set permissions on directories containing the configuration information and persistent data for that broker instance.

A broker instance has the instance name imqbroker by default. To start a broker from the command line with this name and the default configuration, simply use the command

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

To specify an instance name other than the default, use the -name option to the imqbrokerd command. The following command starts a broker with the instance name myBroker:

```
imqbrokerd  -name myBroker
```
Other options are available on the imqbrokerd command line to control various aspects of the broker's operation. See Broker Utility for complete information on the syntax, subcommands, and options of the imqbrokerd command. For a quick summary of this information, enter the following command:

```
imqbrokerd  -help
```
For example, the following command uses the -tty option to send errors and warnings to the command window (standard output):

```
imqbrokerd  -name myBroker  -tty
```
You can also use the -D option on the command line to override the values of properties specified in the broker's instance configuration file (config.properties). The instance configuration file is described under Modifying Configuration Files. The following example sets a broker's imq.jms.max_threads property, raising the maximum number of threads available to the jms connection service to 2000:

```
imqbrokerd  -name myBroker  -Dimq.jms.max_threads=2000
```

Starting Brokers Automatically

Instead of starting a broker explicitly from the command line, you can set it up to start automatically at system startup. How you do this depends on the platform (Solaris, Linux, or Windows) on which you are running the broker:

- Automatic Broker Startup on the Solaris Platforms
- Automatic Broker Startup on the Linux Platform
- Automatic Broker Startup on Windows

Automatic Broker Startup on the Solaris Platforms

The method for enabling automatic startup on the Solaris 10 platforms is different from that for Solaris 9. Both are described below.

Automatic Broker Startup on the Solaris 9 Platform On Solaris 9 operating system, scripts that enable automatic startup are placed in the `/etc/rc*` directory tree during Message Queue installation. To enable the use of these scripts, you must edit the configuration file `imqbrokerd.conf` (located in the `IMQ_HOME/etc/` directory) as follows:

- To start the broker automatically at system startup, set the `AUTOSTART` property to `YES`.
- To have the broker restart automatically after an abnormal exit, set the `RESTART` property to `YES`.
- To set startup command line arguments for the broker, specify one or more values for the `ARGS` property.

To disable automatic broker startup at system startup, edit the configuration file `/etc/imq/imqbrokerd.conf` and set the `AUTOSTART` property to `NO`.

Automatic Broker Startup on the Solaris 10 Platform Rather than using an `rc` file to implement automatic broker startup when a computer reboots, the following procedure makes use of the Solaris 10 Service Management Facility (SMF).

For more information on using the Service Management Facility, please refer to Solaris 10 documentation.

To Implement Automatic Broker Startup on Solaris 10 OS 1. Copy and change permissions on the `mqbroker` startup script.

```
# cp /var/svc/manifest/application/sun/mq/mqbroker /lib/svc/method
```

```
# chmod 555 /lib/svc/method/mqbroker
```

2. Import the `mqbroker` service into the SMF repository.

```
# svccfg import /var/svc/manifest/application/sun/mq/mqbroker.xml
```

3. Verify that the import was successful by checking the state of the `mqbroker` service.

```
# svcs mqbroker
```

Output resembles the following:

```
STATE STIME FMRI
disabled 16:22:50 svc:/application/sun/mq/mqbroker:default
```

The service is initially shown as disabled.

4. Eanable the `mqbroker` service.

```
# svcadm enable svc:/application/sun/mq/mqbroker:default
```

Enabling the mqbroker service will start the imqbrokerd process. A reboot will subsequently restart the broker.

5. Configure the mqbroker service to pass any desired arguments to the imqbrokerd command.

The options/broker_args property is used to pass arguments to imqbrokerd. For example to add -loglevel DEBUGHIGH, do the following:

```
# svccfg
svc:> select svc:/application/sun/mq/mqbroker
svc:/application/sun/mq/mqbroker> setprop options/broker_args="-loglevel
DEBUGHIGH"
svc:/application/sun/mq/mqbroker> exit
```

To Disable Automatic Broker Startup on Solaris 10 OS

1. Disable the mqbroker service.

```
# svcadm disable svc:/application/sun/mq/mqbroker:default
```

A subsequent reboot will not restart the broker.

Automatic Broker Startup on the Linux Platform

On Linux systems, scripts that enable automatic startup are placed in the /etc/rc* directory tree during Message Queue installation. To enable the use of these scripts, you must edit the configuration file imqbrokerd.conf (located in the IMQ_HOME/etc/ directory) as follows:

- To start the broker automatically at system startup, set the AUTOSTART property to YES.

- To have the broker restart automatically after an abnormal exit, set the RESTART property to YES.

- To set startup command line arguments for the broker, specify one or more values for the ARGS property.

To disable automatic broker startup at system startup, edit the configuration file /etc/opt/sun/mq/imqbrokerd.conf and set the AUTOSTART property to NO.

Automatic Broker Startup on Windows

To start a broker automatically at Windows system startup, you must define the broker as a Windows service. The broker will then start at system startup time and run in the background until system shutdown. Consequently, you will not need to use the Message Queue Broker utility (imqbrokerd) unless you want to start an additional broker.

A system can have no more than one broker running as a Windows service. The Windows Task Manager lists such a broker as two executable processes:

- The native Windows service wrapper, imqbrokersvc.exe

- The Java runtime that is running the broker

You can install a broker as a service when you install Message Queue on a Windows system. After installation, you can use the Service Administrator utility (imqsvcadmin) to perform the following operations:

- Add a broker as a Windows service

- Determine the startup options for the broker service

- Disable a broker from running as a Windows service

To pass startup options to the broker, use the `-args` option to the `imqsvcadmin` command. This works the same way as the `imqbrokerd` command's `-D` option, as described under Starting Brokers. Use the Command utility (`imqcmd`) to control broker operations as usual.

See Service Administrator Utility for complete information on the syntax, subcommands, and options of the `imqsvcadmin` command.

Reconfiguring the Broker Service The procedure for reconfiguring a broker installed as a Windows service is as follows:

To Reconfigure a Broker Running as a Windows Service

1. Stop the service:

   a. From the Settings submenu of the Windows Start menu, choose Control Panel.

   b. Open the Administrative Tools control panel.

   c. Run the Services tool by selecting its icon and choosing Open from the File menu or the pop-up context menu, or simply by double-clicking the icon.

   d. Under Services (Local), select the Message Queue Broker service and choose Properties from the Action menu.

      Alternatively, you can right-click on Message Queue Broker and choose Properties from the pop-up context menu, or simply double-click on Message Queue Broker. In either case, the Message Queue Broker Properties dialog box will appear.

   e. Under the General tab in the Properties dialog, click Stop to stop the broker service.

2. Remove the service.

   On the command line, enter the command

   ```
   imqsvcadmin remove
   ```

3. Reinstall the service, specifying different broker startup options with the `-args` option or different Java version arguments with the `-vmargs` option.

   For example, to change the service's host name and port number to `broker1` and `7878`, you could use the command

   ```
   imqsvcadmin install  -args "-name broker1  -port 7878"
   ```

Using an Alternative Java Runtime You can use either the `imqsvcadmin` command's `-javahome` or `-jrehome` option to specify the location of an alternative Java runtime. (You can also specify these options in the Start Parameters field under the General tab in the service's Properties dialog window.)

> **Note:** The Start Parameters field treats the backslash character (\) as an escape character, so you must type it twice when using it as a path delimiter: for example,
>
> ```
> -javahome c:\\j2sdk1.4.0
> ```

Displaying Broker Service Startup Options To determine the startup options for the broker service, use the `imqsvcadmin query` command, as shown in Example 3–1.

Example 3–1   Displaying Broker Service Startup Options

```
imqsvcadmin query

Service Message Queue Broker is installed.
Display Name: Message Queue Broker
Start Type: Automatic
Binary location: C:\Sun\MessageQueue\bin\imqbrokersvc.exe
JavaHome: c:\j2sdk1.4.0
Broker Args: -name broker1 -port 7878
```

Disabling a Broker From Running as a Windows Service To disable a broker from running as a Windows service, use the command

```
imqcmd shutdown bkr
```
to shut down the broker, followed by

```
imqsvcadmin remove
```
to remove the service.

Alternatively, you can use the Windows Services tool, reached via the Administrative Tools control panel, to stop and remove the broker service.

Restart your computer after disabling the broker service.

Troubleshooting Service Startup Problems If you get an error when you try to start a broker as a Windows service, you can view error events that were logged:

To See Logged Service Error Events

1. Open the Windows Administrative Tools control panel.

2. Start the Event Viewer tool.

3. Select the Application event log.

4. Choose Refresh from the Action menu to display any error events.

Deleting a Broker Instance

To delete a broker instance, use the `imqbrokerd` command with the `-remove` option:

```
imqbrokerd [options…]-remove instance
```
For example, if the name of the broker is `myBroker`, the command would be

```
imqbrokerd  -name myBroker  -remove instance
```
The command deletes the entire instance directory for the specified broker.

See Broker Utility for complete information on the syntax, subcommands, and options of the `imqbrokerd` command. For a quick summary of this information, enter the command

```
imqbrokerd  -help
```

Starting Clients

Before starting a client application, obtain information from the application developer about how to set up the system. If you are starting Java client applications, you must

set the `CLASSPATH` variable appropriately and make sure you have the correct `.jar` files installed. The *Open Message Queue Developer's Guide for Java Clients* contains information about generic steps for setting up the system, but your developer may have additional information to provide.

To start a Java client application, use the following command line format:

```
java clientAppName
```
To start a C client application, use the format supplied by the application developer (see "Building and Running C Clients" in *Open Message Queue Developer's Guide for C Clients*).

The application's documentation should provide information on attribute values that the application sets; you may want to override some of these from the command line. You may also want to specify attributes on the command line for any Java client that uses a Java Naming and Directory Interface (JNDI) lookup to find its connection factory. If the lookup returns a connection factory that is older than the application, the connection factory may lack support for more recent attributes. In such cases, Message Queue sets those attributes to default values; if necessary, you can use the command line to override these default values.

To specify attribute values from the command line for a Java application, use the following syntax:

```
java [ [-Dattribute=value] … ]clientAppName
```
The value for *attribute* must be a connection factory administered object attribute, as described in [Administered Object Attribute Reference](#). If there is a space in the value, put quotation marks around the

```
attribute=value
```
part of the command line.

The following example starts a client application named MyMQClient , connecting to a broker on the host `OtherHost` at port `7677`:

```
java -DimqAddressList=mq://OtherHost:7677/jms  MyMQClient
```
The host name and port specified on the command line override any others set by the application itself.

In some cases, you cannot use the command line to specify attribute values. An administrator can set an administered object to allow read access only, or an application developer can code the client application to do so. Communication with the application developer is necessary to understand the best way to start the client program.

4

Configuring a Broker

A broker's configuration is governed by a set of configuration files and by the options passed to the `imqbrokerd` command at startup. This chapter describes the available configuration properties and how to use them to configure a broker.

The chapter contains the following sections:

- Broker Services
- Setting Broker Configuration Properties

For full reference information about broker configuration properties, see Broker Properties Reference.

Broker Services

Broker configuration properties are logically divided into categories that depend on the services or broker components they affect:

- *Connection services* manage the physical connections between a broker and its clients that provide transport for incoming and outgoing messages. For a discussion of properties associated with connection services, see Configuring Connection Services.

- *Message delivery services* route and deliver JMS payload messages, as well as control messages used by the message service to support reliable delivery. For a discussion of properties associated with message delivery services, including physical destinations, see Managing Message Delivery.

- *Persistence services* manage the writing and retrieval of data, such as messages and state information, to and from persistent storage. For a discussion of properties associated with persistence services, see Configuring Persistence Services.

- *Security services* authenticate users connecting to the broker and authorize their actions. For a discussion of properties associated with authentication and authorization services, as well as encryption configuration, see Configuring and Managing Security Services.

- *Clustering services* support the grouping of brokers into a cluster to achieve scalability and availability. For a discussion of properties associated with broker clusters, see Configuring and Managing Broker Clusters.

- *Monitoring services* generate metric and diagnostic information about the broker's performance. For a discussion of properties associated with monitoring and managing a broker, see Monitoring Broker Operations.

Setting Broker Configuration Properties

You can specify a broker's configuration properties in either of two ways:

- Edit the broker's configuration file.

- Supply the property values directly from the command line.

The following sections describe these two methods of configuring a broker.

Modifying Configuration Files
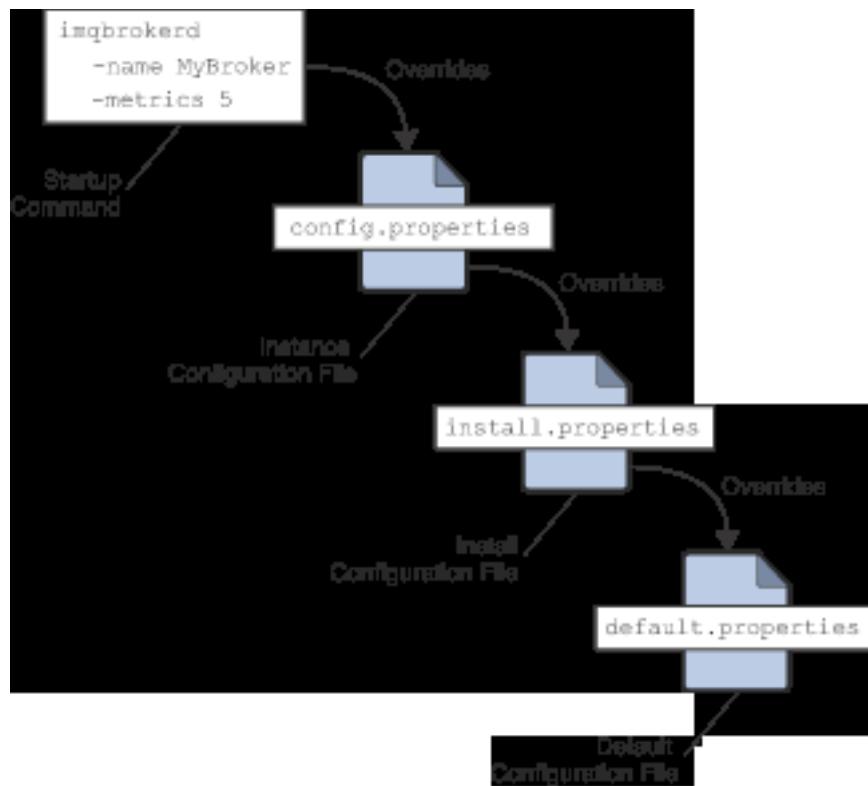
Broker configuration files contain property settings for configuring a broker. Message Queue maintains the following broker configuration files:

- A *default configuration file* (`IMQ_HOME/lib/props/broker/default.properties`) that is loaded on startup. This file is not editable, but you can read it to determine default settings and find the exact names of properties you want to change.

- An *installation configuration file* (`IMQ_ HOME/lib/props/broker/install.properties`) containing any properties specified when Message Queue was installed. This file cannot be edited after installation.

- A separate *instance configuration file* (`IMQ_ VARHOME/instances/`*instanceName*`/props/config.properties`) for each individual broker instance.

In addition, if you connect broker instances in a cluster, you may need to use a *cluster configuration file* (`cluster.properties`) to specify configuration information for the cluster; see Cluster Configuration Properties for more information.

Also, Message Queue makes use of en environment configuration file, `imqenv.conf`, which is used to provide the locations of external files needed by Message Queue, such as the default Java SE location and the locations of database drivers, JAAS login modules, and so forth.

At startup, the broker merges property values from the various configuration files. As shown in Figure 4–1, the files form a hierarchy in which values specified in the instance configuration file override those in the installation configuration file, which in turn override those in the default configuration file. At the top of the hierarchy, you can manually override any property values specified in the configuration files by using command line options to the `imqbrokerd` command.

Figure 4–1   Broker Configuration Files

The first time you run a broker, an instance configuration file is created containing configuration properties for that particular broker instance. The instance configuration file is named `config.properties` and is located in a directory identified by the name of the broker instance to which it belongs:

`IMQ_VARHOME/instances/`*instanceName*`/props/config.properties`

If the file does not yet exist, you must use the `-name` option when starting the broker (see Broker Utility) to specify an instance name that Message Queue can use to create the file.

> **Note:**   The `instances/`*instanceName* directory and the instance configuration file are owned by the user who initially started the corresponding broker instance by using the `imqbrokerd -name` *brokerName* option. The broker instance must always be restarted by that same user.

The instance configuration file is maintained by the broker instance and is modified when you make configuration changes using Message Queue administration utilities. You can also edit an instance configuration file by hand. To do so, you must be the owner of the `instances/`*instanceName* directory or log in as the root user to change the directory's access privileges.

The broker reads its instance configuration file only at startup. To effect any changes to the broker's configuration, you must shut down the broker and then restart it. Property definitions in the `config.properties` file (or any configuration file) use the following syntax:

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

For example, the following entry specifies that the broker will hold up to 50,000 messages in memory and persistent storage before rejecting additional messages:

```
imq.system.max_count=50000
```
The following entry specifies that a new log file will be created once a day (every 86,400 seconds):

```
imq.log.file.rolloversecs=86400
```
See Broker Services and Broker Properties Reference for information on the available broker configuration properties and their default values.

Setting Configuration Properties from the Command Line

You can enter broker configuration properties from the command line when you start a broker, or afterward.

At startup time, you use the Broker utility (imqbrokerd) to start a broker instance. Using the command's -D option, you can specify any broker configuration property and its value; see Starting Brokers and Broker Utility for more information. If you start the broker as a Windows service, using the Service Administrator utility (imqsvcadmin), you use the -args option to specify startup configuration properties; see Service Administrator Utility.

You can also change certain broker configuration properties while a broker is running. To modify the configuration of a running broker, you use the Command utility's imqcmd update bkr command; see Updating Broker Properties and Broker Management.

5

Managing a Broker

This chapter explains how to use the Message Queue Command utility (`imqcmd`) to manage a broker. The chapter has the following sections:

- Command Utility Preliminaries
- Using the Command Utility
- Managing Brokers

This chapter does not cover all topics related to managing a broker. Additional topics are covered in the following separate chapters:

- For information on configuring and managing connection services, see Configuring and Managing Connection Services.

- For information on managing message delivery services, including how to create, display, update, and destroy physical destinations, see Managing Message Delivery.

- For information on configuring and managing persistence services, for both flat-file and JDBC-based data stores, see Configuring Persistence Services.

- For information about setting up security for the broker, such as user authentication, access control, encryption, and password files, see Configuring and Managing Security Services.

- For information on configuring and managing clustering services, for both conventional and enhanced broker clusters, see Configuring and Managing Broker Clusters.

- For information about monitoring a broker, see Monitoring Broker Operations.

Command Utility Preliminaries

Before using the Command utility to manage a broker, you must do the following:

- Start the broker using the `imqbrokerd` command. You cannot use the Command utility subcommands l until a broker is running.

- Determine whether you want to set up a Message Queue administrative user or use the default account. You must specify a user name and password to use all Command utility subcommands (except to display command help and version information).

  When you install Message Queue, a default flat-file user repository is installed. The repository is shipped with two default entries: an administrative user and a guest user. If you are testing Message Queue, you can use the default user name and password (`admin/admin`) to run the Command utility.

If you are setting up a production system, you must set up authentication and authorization for administrative users. See Configuring and Managing Security Services for information on setting up a file-based user repository or configuring the use of an LDAP directory server. In a production environment, it is a good security practice to use a nondefault user name and password.

- If you want to use a secure connection to the broker, set up and enable the ssladmin service on the target broker instance, For more information, see Message Encryption.

Using the Command Utility

The Message Queue Command utility (`imqcmd`) enables you to manage the broker and its services interactively from the command line. See Command Utility for general reference information about the syntax, subcommands, and options of the `imqcmd` command, and Broker Properties Reference for specific information on the configuration properties used to specify broker behavior.

Specifying the User Name and Password

Because each `imqcmd` subcommand is authenticated against the user repository, it requires a user name and password. The only exceptions are commands that use the `-h` or `-H` option to display help, and those that use the `-v` option to display the product version.

Use the `-u` option to specify an administrative user name. For example, the following command displays information about the default broker:

```
imqcmd query bkr  -u admin
```
If you omit the user name, the command will prompt you for it.

> **Note:** For simplicity, the examples in this chapter use the default user name `admin` as the argument to the `-u` option. In a real-life production environment, you would use a custom user name.

Specify the password using one of the following methods:

- Create a password file and enter the password into that file as the value of the `imq.imqcmd.password` property. On the command line, use the `-passfile` option to provide the name of the password file.

- Let the `imqcmd` command prompt you for the password.

> **Note:** In previous versions of Message Queue, you could use the `-p` option to specify a password on the `imqcmd` command line. As of Message Queue 4.0, this option is deprecated and is no longer supported; you must instead use one of the methods listed above.

Specifying the Broker Name and Port

Most `imqcmd` subcommands use the `-b` option to specify the host name and the port mapper port number of the broker to which the command applies:

```
-b hostName:portNumber
```

If no broker is specified, the command applies by default to a broker running on the local host (`localhost`) at port mapper port number `7676`. See "Port Mapper" for more information.

To issue a command to a broker that is running on a remote host, listening on a non-default port, or both, you must use the `-b` option to specify the host and port explicitly.

Literal IP addresses as host names: You can use a literal IPv4 or IPv6 address as a host name. If you use a literal IPv6 address, its format must conform to RFC2732 (`http://www.ietf.org/rfc/rfc2732.txt`), *Format for Literal IPv6 Addresses in URL's.*

Displaying the Product Version

To display the Message Queue product version, use the `-v` option. For example:

```
imqcmd  -v
```
If you enter an `imqcmd` command line containing the `-v` option in addition to a subcommand or other options, the Command utility processes only the `-v` option. All other items on the command line are ignored.

Displaying Help

To display help on the `imqcmd` command, use the `-h` or `-H` option, and do not use a subcommand. You cannot get help about specific subcommands.

For example, the following command displays help about `imqcmd`:

```
imqcmd  -H
```
If you enter an `imqcmd` command line containing the `-h` or `-H` option in addition to a subcommand or other options, the Command utility processes only the `-h` or `-H` option. All other items on the command line are ignored.

Examples

The examples in this section illustrate how to use the `imqcmd` command.

The following example lists the properties of the broker running on host `localhost` at port `7676`, so the `-b` option is unnecessary:

```
imqcmd query bkr  -u admin
```
The command uses the default administrative user name (`admin`) and omits the password, so that the command will prompt for it.

The following example lists the properties of the broker running on the host `myserver` at port `1564`. The user name is `aladdin`:

```
imqcmd query bkr  -b myserver:1564  -u aladdin
```
(For this command to work, the user repository would need to be updated to add the user name `aladdin` to the `admin` group.)

The following example lists the properties of the broker running on `localhost` at port `7676`. The initial timeout for the command is set to 20 seconds and the number of retries after timeout is set to 7. The user's password is in a password file called `myPassfile`, located in the current directory at the time the command is invoked.

```
imqcmd query bkr  -u admin  -passfile myPassfile  -rtm 20  -rtr 7
```
For a secure connection to the broker, these examples could include the `-secure` option. This option causes the Command utility to use the `ssladmin` service if that service has been configured and started.

Managing Brokers

This section describes how to use Command utility subcommands to perform the following broker management tasks:

- Shutting Down and Restarting a Broker
- Quiescing a Broker
- Pausing and Resuming a Broker
- Updating Broker Properties
- Viewing Broker Information

In addition to using the subcommands described in the following sections, `imqcmd` allows you to set system properties using the `-D` option. This is useful for setting or overriding connection factory properties or connection-related Java system properties.

For example, the following command changes the default value of `imqSSLIsHostTrusted`:

```
imqcmd list svc -secure -DimqSSLIsHostTrusted=true
```

The following command specifies the password file and the password used for the SSL trust store that is used by the `imqcmd` command:

```
imqcmd list svc -secure -DJavax.net.ssl.trustStore=/tmp/MyTruststore
          -Djavax.net.ssl.trustStorePassword=MyTrustword
```

Shutting Down and Restarting a Broker

The subcommand `imqcmd shutdown bkr` shuts down a broker:

```
imqcmd shutdown bkr  [-b hostName:portNumber]
[-time nSeconds]
[-nofailover]
```

The broker stops accepting new connections and messages, completes delivery of existing messages, and terminates the broker process.

The `-time` option, if present, specifies the interval, in seconds, to wait before shutting down the broker. For example, the following command delays 90 seconds and then shuts down the broker running on host `wolfgang` at port `1756`:

```
imqcmd shutdown bkr  -b wolfgang:1756  -time 90  -u admin
```

The broker will not block, but will return immediately from the delayed shutdown request. During the shutdown interval, the broker will not accept any new `jms` connections; `admin` connections will be accepted, and existing `jms` connections will continue to operate. If the broker belongs to an enhanced broker cluster, it will not attempt to take over for any other broker during the shutdown interval.

If the broker is part of an enhanced broker cluster (see "Enhanced Clusters" in *Open Message Queue Technical Overview*), another broker in the cluster will ordinarily attempt to take over its persistent data on shutdown; the `-nofailover` option to the `imqcmd shutdown bkr` subcommand suppresses this behavior. Conversely, you can use the `imqcmd takeover bkr` subcommand to force such a takeover manually (for instance, if the takeover broker were to fail before completing the takeover process); see Preventing or Forcing Broker Failover for more information.

> **Note:** The `imqcmd takeover bkr` subcommand is intended only for use in failed-takeover situations. You should use it only as a last resort, and not as a general way of forcibly taking over a running broker.

To shut down and restart a broker, use the subcommand `imqcmd restart bkr`:

```
imqcmd restart bkr  [-b hostName:portNumber]
```
This shuts down the broker and then restarts it using the same options that were specified when it was first started. To choose different options, shut down the broker with `imqcmd shutdown bkr` and then start it again with the Broker utility (`imqbrokerd`), specifying the options you want.

Quiescing a Broker

The subcommand `imqcmd quiesce bkr` quiesces a broker, causing it to refuse any new client connections while continuing to service old ones:

```
imqcmd quiesce bkr  [-b hostName:portNumber]
```
If the broker is part of an enhanced broker cluster, this allows its operations to wind down normally without triggering a takeover by another broker, for instance in preparation for shutting it down administratively for upgrade or similar purposes. For example, the following command quiesces the broker running on host `hastings` at port `1066`:

```
imqcmd quiesce bkr  -b hastings:1066  -u admin
```
To reverse the process and return the broker to normal operation, use the `imqcmd unquiesce bkr` subcommand:

```
imqcmd unquiesce bkr  [-b hostName:portNumber]
```
For example, the following command unquiesces the broker that was quiesced in the preceding example:

```
imqcmd unquiesce bkr  -b hastings:1066  -u admin
```

Pausing and Resuming a Broker

The subcommand `imqcmd pause bkr` pauses a broker, suspending its connection service threads and causing it to stop listening on the connection ports:

```
imqcmd pause bkr  [-b hostName:portNumber]
```
For example, the following command pauses the broker running on host `myhost` at port `1588`:

```
imqcmd pause bkr  -b myhost:1588  -u admin
```
Because its connection service threads are suspended, a paused broker is unable to accept new connections, receive messages, or dispatch messages. The `admin` connection service is *not* suspended, allowing you to continue performing administrative tasks needed to regulate the flow of messages to the broker. Pausing a broker also does not suspend the `cluster` connection service; however, since message delivery within a cluster depends on the delivery functions performed by the different brokers in the cluster, pausing a broker in a cluster may result in a slowing of some message traffic.

You can also pause individual connection services and physical destinations. For more information, see Pausing and Resuming a Connection Service and Pausing and Resuming a Physical Destination.

The subcommand `imqcmd resume bkr` reactivates a broker's service threads, causing it to resume listening on the ports:

```
imqcmd resume bkr  [-b hostName:portNumber]
```
For example, the following command resumes the default broker (host `localhost` at port `7676`):

```
imqcmd resume bkr  -u admin
```

Updating Broker Properties

The subcommand `imqcmd update bkr` can be used to change the values of a subset of broker properties for the default broker (or for the broker at a specified host and port):

```
imqcmd update bkr [-b hostName:portNumber]
-o property1=value1  [ [-o property2=value2] … ]
```
For example, the following command turns off the auto-creation of queue destinations for the default broker:

```
imqcmd update bkr  -o imq.autocreate.queue=false  -u admin
```
You can use `imqcmd update bkr` to update any of the following broker properties:

- `imq.autocreate.queue`

- `imq.autocreate.topic`

- `imq.autocreate.queue.maxNumActiveConsumers`

- `imq.autocreate.queue.maxNumBackupConsumers`

- `imq.cluster.url`

- `imq.destination.DMQ.truncateBody`

- `imq.destination.logDeadMsgs`

- `imq.log.level`

- `imq.log.file.rolloversecs`

- `imq.log.file.rolloverbytes`

- `imq.system.max_count`

- `imq.system.max_size`

- `imq.message.max_size`

- `imq.portmapper.port`

See Broker Properties Reference for detailed information about these properties.

Viewing Broker Information

To display information about a single broker, use the `imqcmd query bkr` subcommand:

```
imqcmd query bkr  -b hostName:portNumber
```
This lists the current settings of the broker's properties, as shown in Example 5–1.

Example 5–1   Broker Information Listing

```
Querying the broker specified by:
-----------------------
Host        Primary Port
-----------------------
```

```
localHost    7676

Version                                           5.0
Instance Name                                     imqbroker
Broker ID                                         mybroker
Primary Port                                      7676
Broker is Embedded                                false
Instance Configuration/Data Root Directory        /var/imq

Current Number of Messages in System              0
Current Total Message Bytes in System             0

Current Number of Messages in Dead Message Queue  0
Current Total Message Bytes in Dead Message Queue 0
Log Dead Messages                                 true
Truncate Message Body in Dead Message Queue       false

Max Number of Messages in System                  unlimited (-1)
Max Total Message Bytes in System                 unlimited (-1)
Max Message Size                                  70m

Auto Create Queues                                true
Auto Create Topics                                true
Auto Created Queue Max Number of Active Consumers  1
Auto Created Queue Max Number of Backup Consumers  0

Cluster ID                                        myClusterID
Cluster Is Highly Available                       true
Cluster Broker List (active)
Cluster Broker List (configured)
Cluster Master Broker
Cluster URL

Log Level                                         INFO
Log Rollover Interval (seconds)                   604800
Log Rollover Size (bytes)                         unlimited (-1)
```

The `imqcmd metrics bkr` subcommand displays detailed metric information about a broker's operation:

```
imqcmd metrics bkr  [-b hostName:portNumber]
[-m metricType]
[-int interval]
[-msp numSamples]
```

The `-m` option specifies the type of metric information to display:

- `ttl` *(default)*: Messages and packets flowing into and out of the broker

- `rts`: Rate of flow of messages and packets into and out of the broker per second

- `cxn`: Connections, virtual memory heap, and threads

The `-int` and `-msp` options specify, respectively, the interval (in seconds) at which to display the metrics and the number of samples to display in the output. The default values are 5 seconds and an unlimited number of samples.

For example, the following command displays the rate of message flow into and out of the default broker (host `localhost` at port `7676`) at 10-second intervals:

```
imqcmd metrics bkr  -m rts  -int 10  -u admin
```
Example 5–2 shows an example of the resulting output.

Example 5–2   Broker Metrics Listing

```
--------------------------------------------------------
Msgs/sec   Msg Bytes/sec    Pkts/sec    Pkt Bytes/sec
In    Out     In      Out     In    Out     In      Out
--------------------------------------------------------
0     0       27      56      0     0       38      66
10    0       7365    56      10    10      7457    1132
0     0       27      56      0     0       38      73
0     10      27      7402    10    20      1400    8459
0     0       27      56      0     0       38      73
```

For a more detailed description of the data gathered and reported by the broker, see Brokerwide Metrics.

For brokers belonging to a broker cluster, the imqcmd list bkr subcommand displays information about the configuration of the cluster; see Displaying a Cluster Configuration for more information.

6

Configuring and Managing Connection Services
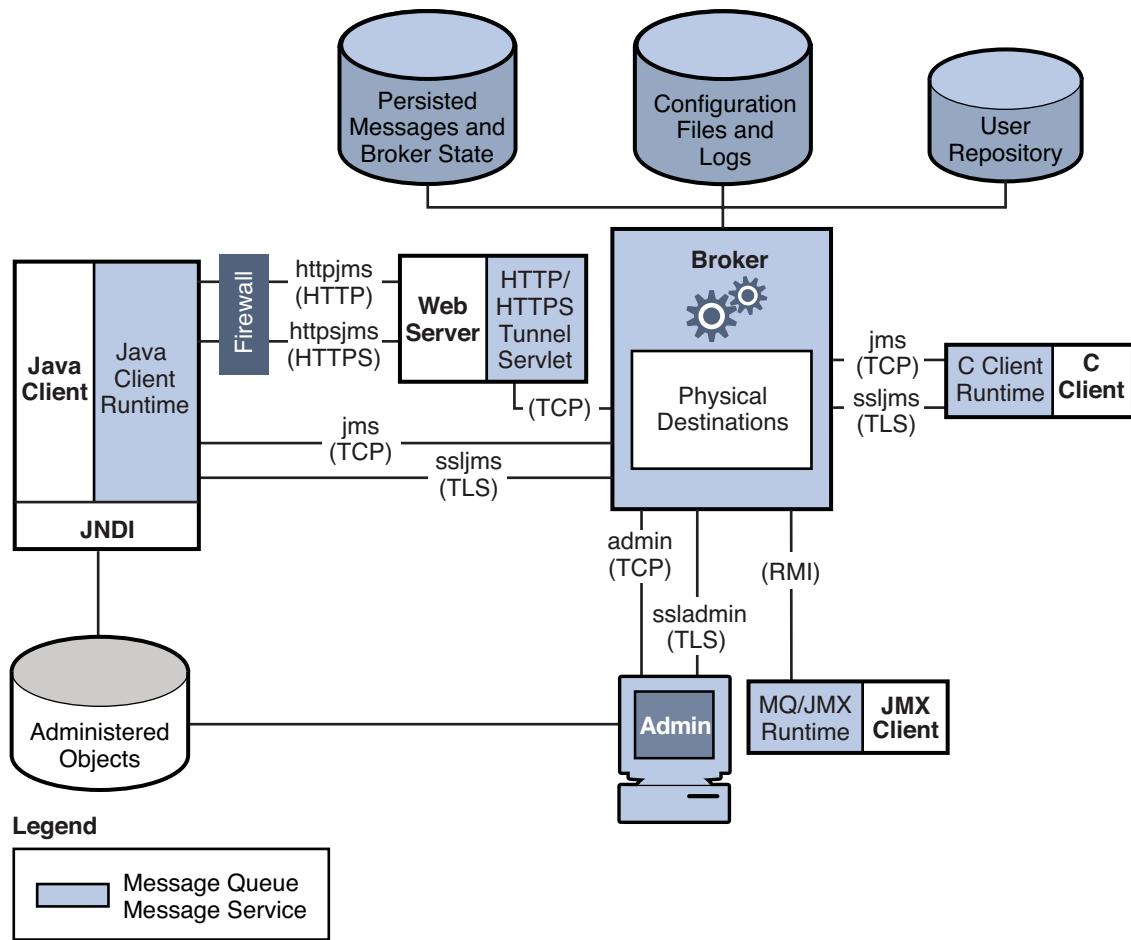
Message Queue offers various *connection services* using a variety of transport protocols for connecting both application and administrative clients to a broker. This chapter describes how to configure and manage these services and the connections they support:

- Configuring Connection Services
- Managing Connection Services
- Managing Connections

Configuring Connection Services

Broker configuration properties related to connection services are listed under Connection Properties.

Figure 6–1 shows the connection services provided by the Message Queue broker.

Figure 6–1   Message Queue Connection Services

These connection services are distinguished by two characteristics, as shown in Table 6–1:

- The *service type* specifies whether the service provides JMS message delivery (NORMAL) or Message Queue administration services (ADMIN).

- The *protocol type* specifies the underlying transport protocol.

Table 6–1   Message Queue Connection Service Characteristics

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

By setting a broker's imq.service.activelist property, you can configure it to run any or all of these connection services. The value of this property is a list of connection services to be activated when the broker is started up; if the property is not specified explicitly, the jms and admin services will be activated by default.

Each connection service also supports specific authentication and authorization features; see Introduction to Security Services for more information.

> **Note:** There is also a special `cluster` connection service, used internally by the brokers within a broker cluster to exchange information about the cluster's configuration and state. This service is not intended for use by clients communicating with a broker. See Configuring and Managing Broker Clusters for more information about broker clusters.
>
> Also there are two JMX connectors, `jmxrmi` and `ssljmxrmi`, that support JMX-based administration. These JMX connectors are very similar to the connection services in Table 6–1, above, and are used by JMX clients to establish a connection to the broker's MBean server. For more information, see JMX Connection Infrastructure.

Port Mapper

Each connection service is available at a particular port, specified by host name (or IP address) and port number. You can explicitly specify a static port number for a service or have the broker's *Port Mapper* assign one dynamically. The Port Mapper itself resides at the broker's *primary port,* which is normally located at the standard port number `7676`. (If necessary, you can use the broker configuration property `imq.portmapper.port` to override this with a different port number.) By default, each connection service registers itself with the Port Mapper when it starts up. When a client creates a connection to the broker, the Message Queue client runtime first contacts the Port Mapper, requesting a port number for the desired connection service.

Alternatively, you can override the Port Mapper and explicitly assign a static port number to a connection service, using the `imq.`*serviceName*.*protocolType*`.port` configuration property (where *serviceName* and *protocolType* identify the specific connection service, as shown in Table 6–1). (Only the `jms`, `ssljms`, `admin`, and `ssladmin` connection services can be configured this way; the `httpjms` and `httpsjms` services use different configuration properties, described in HTTP/HTTPS Support). Static ports are generally used only in special situations, however, such as in making connections through a firewall (see Connecting Through a Firewall), and are not recommended for general use.

> **Note:** In cases where two or more hosts are available (such as when more than one network interface card is installed in a computer), you can use broker properties to specify which host the connection services should bind to. The `imq.hostname` property designates a single default host for all connection services; this can then be overridden, if necessary, with `imq.`*serviceName*.*protocolType*.`hostname` (for the `jms`, `ssljms`, `admin`, or `ssladmin` service) or `imq.portmapper.hostname` (for the Port Mapper itself).

When multiple Port Mapper requests are received concurrently, they are stored in an operating system backlog while awaiting action. The `imq.portmapper.backlog` property specifies the maximum number of such backlogged requests. When this limit is exceeded, any further requests will be rejected until the backlog is reduced.

Thread Pool Management

Each connection service is multithreaded, supporting multiple connections. The threads needed for these connections are maintained by the broker in a separate *thread pool* for each service. As threads are needed by a connection, they are added to the thread pool for the service supporting that connection.

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

- In the *dedicated model,* each connection to the broker requires two threads: one for incoming and one for outgoing messages. This limits the number of connections that can be supported, but provides higher performance.

- In the *shared model,* connections are processed by a shared thread when sending or receiving messages. Because each connection does not require dedicated threads, this model increases the number of possible connections, but at the cost of lower performance because of the additional overhead needed for thread management.

The broker's imq.*serviceName.* threadpool_model property specifies which of the two models to use for a given connection service. This property takes either of two string values: dedicated or shared. If you don't set the property explicitly, dedicated is assumed by default.

You can also set the broker properties imq.*serviceName.* min_threads and imq.*serviceName.* max_threads to specify a minimum and maximum number of threads in a service's thread pool. When the number of available threads exceeds the specified minimum threshold, Message Queue will shut down threads as they become free until the minimum is reached again, thereby saving on memory resources. Under heavy loads, the number of threads might increase until the pool's maximum number is reached; at this point, new connections are rejected until a thread becomes available.

The shared threading model uses *distributor threads* to assign threads to active connections. The broker property imq.shared.connectionMonitor_limit specifies the maximum number of connections that can be monitored by a single distributor thread. The smaller the value of this property, the faster threads can be assigned to connections. The imq.ping.interval property specifies the time interval, in seconds, at which the broker will periodically test ("ping") a connection to verify that it is still active, allowing connection failures to be detected preemptively before an attempted message transmission fails.

Managing Connection Services

Message Queue brokers support connections from both application clients and administrative clients. See Configuring Connection Services for a description of the available connection services. The Command utility provides subcommands that you can use for managing both connection services as a whole and individual services; to apply a subcommand to a particular service, use the -n option to specify one of the names listed in the "Service Name" column of Table 6–1. Subcommands are available for the following connection service management tasks:

- Pausing and Resuming a Connection Service

- Updating Connection Service Properties

- Viewing Connection Service Information

Pausing and Resuming a Connection Service

Pausing a connection service has the following effects:

■ The broker stops accepting new client connections on the paused service. If a Message Queue client attempts to open a new connection, it will get an exception.

■ All existing connections on the paused service are kept alive, but the broker suspends all message processing on such connections until the service is resumed. (For example, if a client attempts to send a message, the send method will block until the service is resumed.)

■ The message delivery state of any messages already received by the broker is maintained. (For example, transactions are not disrupted and message delivery will resume when the service is resumed.)

The admin connection service can never be paused; to pause and resume any other service, use the subcommands imqcmd pause svc and imqcmd resume svc. The syntax of the imqcmd pause svc subcommand is as follows:

```
imqcmd pause svc   -n serviceName
                      [-b hostName:portNumber]
```

For example, the following command pauses the httpjms service running on the default broker (host localhost at port 7676):

```
imqcmd pause svc  -n httpjms  -u admin
```

The imqcmd resume svc subcommand resumes operation of a connection service following a pause:

```
imqcmd resume svc   -n serviceName
                       [-b hostName:portNumber]
```

Updating Connection Service Properties

You can use the imqcmd update svc subcommand to change the value of one or more of the service properties listed in Table 6–2. See Connection Properties for a description of these properties.

Table 6–2   Connection Service Properties Updated by Command Utility

Property	Description
port	Port assigned to the service to be updated (does not apply to httpjms or httpsjms)
	A value of 0 means the port is dynamically allocated by the Port Mapper.
minThreads	Minimum number of threads assigned to the service
maxThreads	Maximum number of threads assigned to the service

The imqcmd update svc subcommand has the following syntax:

```
imqcmd update svc   -n serviceName
                       [-b hostName:portNumber]
 -o property1=value1  [[-o property2=value2]…]
```

For example, the following command changes the minimum number of threads assigned to the jms connection service on the default broker (host localhost at port 7676) to 20:

```
imqcmd update svc  -o minThreads=20  -u admin
```

Viewing Connection Service Information

To list the connection services available on a broker, use the imqcmd list svc subcommand:

```
imqcmd list svc  [-b hostName:portNumber]
```
For example, the following command lists all services on the default broker (host `localhost` at port `7676`):

```
imqcmd list svc  -u admin
```
Example 6–1 shows an example of the resulting output.

Example 6–1   Connection Services Listing

```
------------------------------------------------
Service Name    Port Number        Service State
------------------------------------------------
admin           41844 (dynamic)    RUNNING
httpjms         -                  UNKNOWN
httpsjms        -                  UNKNOWN
jms             41843 (dynamic)    RUNNING
ssladmin        dynamic            UNKNOWN
ssljms          dynamic            UNKNOWN
```

The `imqcmd query svc` subcommand displays information about a single connection service:

```
imqcmd query svc   -n serviceName
[-b hostName:portNumber]
```
For example, the following command displays information about the `jms` connection service on the default broker (host `localhost` at port `7676`):

```
imqcmd query svc  -n jms  -u admin
```
Example 6–2 shows an example of the resulting output.

Example 6–2   Connection Service Information Listing

```
Service Name                       jms
Service State                      RUNNING
Port Number                        60920 (dynamic)

Current Number of Allocated Threads  0
Current Number of Connections        0

Min Number of Threads              10
Max Number of Threads              1000
```

To display metrics information about a connection service, use the `imqcmd metrics svc` subcommand:

```
imqcmd metrics svc   -n serviceName
                        [-b hostName:portNumber]
[-m metricType]
[-int interval]
[-msp numSamples]
```
The `-m` option specifies the type of metric information to display:

- `ttl` *(default)*: Messages and packets flowing into and out of the broker by way of the specified connection service

- `rts`: Rate of flow of messages and packets into and out of the broker per second by way of the specified connection service

■   cxn: Connections, virtual memory heap, and threads

The -int and -msp options specify, respectively, the interval (in seconds) at which to display the metrics and the number of samples to display in the output. The default values are 5 seconds and an unlimited number of samples.

For example, the following command displays cumulative totals for messages and packets handled by the default broker (host localhost at port 7676) by way of the jms connection service:

```
imqcmd metrics svc  -n jms  -m ttl  -u admin
```
Example 6–3 shows an example of the resulting output.

Example 6–3   Connection Service Metrics Listing

```
-------------------------------------------------
  Msgs       Msg Bytes      Pkts       Pkt Bytes
In   Out    In     Out    In   Out    In      Out
-------------------------------------------------
164  100  120704  73600  282  383  135967  102127
657  100  483552  73600  775  876  498815  149948
```

For a more detailed description of the use of the Command utility to report connection service metrics, see Connection Service Metrics.

Managing Connections

The Command utility's list cxn and query cxn subcommands display information about individual connections. The subcommand imqcmd list cxn lists all connections for a specified connection service:

```
imqcmd list cxn  [-svn serviceName]
[-b hostName:portNumber]
```
If no service name is specified, all connections are listed. For example, the following command lists all connections on the default broker (host localhost at port 7676):

```
imqcmd list cxn  -u admin
```
Example 6–4 shows an example of the resulting output.

Example 6–4   Broker Connections Listing

```
Listing all the connections on the broker specified by:
----------------------------------
Host                 Primary Port
----------------------------------
localhost            7676


------------------------------------------------------------------------
Connection ID        User   Service  Producers Consumers  Host
------------------------------------------------------------------------
1964412264455443200  guest  jms      0         1          127.0.0.1
1964412264493829311  admin  admin    1         1          127.0.0.1

Successfully listed connections.
```

To display detailed information about a single connection, obtain the connection identifier from imqcmd list cxn and pass it to the imqcmd query cxn subcommand:

```
imqcmd query cxn   -n connectionID
                     [-b hostName:portNumber]
```
For example, the command

```
imqcmd query cxn  -n 421085509902214374  -u admin
```
produces output like that shown in Example 6–5.

Example 6–5   Connection Information Listing

```
Connection ID      421085509902214374
User               guest
Service            jms
Producers          0
Consumers          1
Host               111.22.333.444
Port               60953
Client ID
Client Platform
```

The `imqcmd destroy cxn` subcommand destroys a connection:

```
imqcmd destroy cxn   -n connectionID
                     [-b hostName:portNumber]
```
For example, the command

```
imqcmd destroy cxn  -n 421085509902214374   -u admin
```
destroys the connection shown in Example 6–5.

7

Managing Message Delivery

A Message Queue message is routed to its consumer clients by way of a physical destination on a message broker. The broker manages the memory and persistent storage associated with the physical destination and configures its behavior. The broker also manages memory at a system-wide level, to assure that sufficient resources are available to support all destinations.

Message delivery also involves the maintenance of state information needed by the broker to route messages to consumers and to track acknowledgements and transactions.

This chapter provides information needed to manage message delivery, and includes the following topics:

- Configuring and Managing Physical Destinations
- Managing Broker System-Wide Memory
- Managing Durable Subscriptions
- Managing Transactions

Configuring and Managing Physical Destinations

This section describes how to use the Message Queue Command utility (`imqcmd`) to manage physical destinations. It includes discussion of a specialized physical destination managed by the broker, the dead message queue, whose properties differ somewhat from those of other destinations.

> **Note:**   In a broker cluster, you create a physical destination on one broker and the cluster propagates it to all the others. Because the brokers cooperate to route messages across the cluster, client applications can consume messages from destinations on any broker in the cluster. However the persistence and acknowledgment of a message is managed only by the broker to which a message was originally produced.

This section covers the following topics regarding the management of physical destinations:

- Command Utility Subcommands for Physical Destination Management
- Creating and Destroying Physical Destinations
- Pausing and Resuming a Physical Destination

- [Purging a Physical Destination](#)
- [Updating Physical Destination Properties](#)
- [Viewing Physical Destination Information](#)
- [Managing Physical Destination Disk Utilization](#)
- [Using the Dead Message Queue](#)

> **Note:** For provider independence and portability, client applications typically use destination administered objects to interact with physical destinations. Managing Administered Objects describes how to configure such administered objects for use by client applications. For a general conceptual introduction to physical destinations, see the *Open Message Queue Technical Overview*.

Command Utility Subcommands for Physical Destination Management

The Message Queue Command utility (`imqcmd`) enables you to manage physical destinations interactively from the command line. See Command Line Reference for general reference information about the syntax, subcommands, and options of the `imqcmd` command, and Physical Destination Property Reference for specific information on the configuration properties used to specify physical destination behavior.

Table 7–1 lists the `imqcmd` subcommands for physical destination management. For full reference information about these subcommands, see Table 16–7.

Table 7–1   Physical Destination Subcommands for the Command Utility

Subcommand	Description
`create dst`	Create physical destination
`destroy dst`	Destroy physical destination
`pause dst`	Pause message delivery for physical destination
`resume dst`	Resume message delivery for physical destination
`purge dst`	Purge all messages from physical destination
`compact dst`	Compact physical destination
`update dst`	Set physical destination properties
`list dst`	List physical destinations
`query dst`	List physical destination property values
`metrics dst`	Display physical destination metrics

Creating and Destroying Physical Destinations

The subcommand `imqcmd create dst` creates a new physical destination:

```
imqcmd create dst  -t destType  -n destName
                   [ [-o property=value] … ]
```

You supply the destination type (`q` for a queue or `t` for a topic) and the name of the destination.

Naming Destinations

Destination names must conform to the rules described below for queue and topic destinations.

Supported Queue Destination Names Queue destination names must conform to the following rules:

- It must contain only alphabetic characters (A-Z, a-z), digit characters (0-9), underscores (_), and dollar signs ($).

- It must not contain spaces.

- It must begin with an alphabetic character (A-Z, a-z), an underscore (_), or a dollar sign ($).

- It must not begin with the characters mq.

For example, the following command creates a queue destination named XQueue:

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

Supported Topic Destination Names Topic destination names must conform to the same rules as queue destinations, as specified in Supported Queue Destination Names, except that Message Queue also supports, in addition, topic destination names that include wildcard characters, representing multiple destinations. These symbolic names allow publishers to publish messages to multiple topics and subscribers to consume messages from multiple topics. Using symbolic names, you can create destinations, as needed, consistent with the wildcard naming scheme. Publishers and subscribers automatically publish to and consume from any added destinations that match the symbolic names. (Wildcard topic subscribers are more common than publishers.)

The format of a symbolic topic destination name consists of multiple segments, in which wildcard characters (*, **, >) can represent one or more segments of the name. For example, suppose you have a topic destination naming scheme as follows:

size.color.shape

where the topic name segments can have the following values:

- *size*: large, medium, small, ...

- *color*: red, green, blue, ...

- *shape*: circle, triangle, square, ...

Message Queue supports the following wildcard characters:

- * matches a single segment

- ** matches one or more segments

- > matches any number of segments and must not be proceeded by '.'

You can therefore indicate multiple topic destinations as follows:

large.*.circle would represent:

```
large.red.circle
large.green.circle
...
```

**.square would represent all names ending in .square, for example:

```
small.green.square
medium.blue.square
...
```

small> would represent the destination small and all destination names starting with small., for example:

```
small
small.blue.circle
small.red.square
...
```

It would not, for example, match smallyellow.

To use this multiple destination feature, you create topic destinations using a naming scheme similar to that described above. For example, the following command creates a topic destination named large.green.circle:

```
imqcmd create dst  -t t  -n large.green.circle
```
Client applications can then create wildcard publishers or wildcard consumers using symbolic destination names.

> **Note:**   Oracle recommends storing and looking up Topic objects in JNDI. It is the topic name which contains the wildcard, not the JNDI name

For example:

Example 7–1   Wildcard Publisher

```
...
String topicName = "large.*.circle";
Topic t = session.createTopic(topicName)
TopicPublisher myPublisher = mySession.createPublisher(t)
myPublisher.send(myMessage);
...
```

In this example, the broker will place a copy of the message in any destination that matches the symbolic name large.*.circle

Example 7–2   Wildcard Subscriber

```
...
String topicName = "**.square";
Topic t = session.createTopic(topicName);
TopicSubscriber mySubscriber = mySession.createSubscriber(t);
Message m = mySubscriber.receive();
...
```

In this example, a subscriber will be created if there is at least one destination that matches the symbolic name **.square and will receive messages from all destinations that match that symbolic name. If there are no destinations matching the symbolic name, the subscriber will not be registered with the broker until such a destination exists.

If you create additional destinations that match a symbolic name, then wildcard publishers created using that symbolic name will subsequently publish to that destination and wildcard subscribers created using that symbolic name will subsequently receive messages from that destination.

In addition, Message Queue administration tools, in addition to reporting the total number of publishers (producers) and subscribers (consumers) for a topic destination, will also report the number of publishers that are wildcard publishers (including their corresponding symbolic destination names) and the number of subscribers that are wildcard subscribers (including their symbolic destination names), if any. See Viewing Physical Destination Information.

Setting Property Values

The `imqcmd create dst` command may also optionally include any property values you wish to set for the destination, specified with the `-o` option. For example, the following command creates a topic destination named `hotTopic` with a maximum message length of 5000 bytes:

```
imqcmd create dst  -t t  -n hotTopic  -o maxBytesPerMsg=5000
```

See Physical Destination Property Reference for reference information about the physical destination properties that can be set with this option. (For auto-created destinations, you set default property values in the broker's instance configuration file; see Table 17–3 for information on these properties.)

Destroying Destinations

To destroy a physical destination, use the `imqcmd destroy dst` subcommand:

```
imqcmd destroy dest  -t destType  -n destName
```
This purges all messages at the specified destination and removes it from the broker; the operation is not reversible.

For example, the following command destroys the queue destination named `curlyQueue`:

```
imqcmd destroy dest  -t q  -n curlyQueue  -u admin
```

Note: You cannot destroy the dead message queue.

Pausing and Resuming a Physical Destination

Pausing a physical destination temporarily suspends the delivery of messages from producers to the destination, from the destination to consumers, or both. This can be useful, for instance, to prevent destinations from being overwhelmed when messages are being produced much faster than they are consumed. You must also pause a physical destination before compacting it (see Managing Physical Destination Disk Utilization).

To pause the delivery of messages to or from a physical destination, use the `imqcmd pause dst` subcommand:

```
imqcmd pause dest  [-t destType  -n destName]
[-pst pauseType]
```
If you omit the destination type and name (`-t` and `-n` options), all physical destinations will be paused. The pause type (`-pst`) specifies what type of message delivery to pause:

PRODUCERS
Pause delivery from message producers to the destination

CONSUMERS
Pause delivery from the destination to message consumers

ALL

Pause all message delivery (both producers and consumers)

If no pause type is specified, all message delivery will be paused.

For example, the following command pauses delivery from message producers to the queue destination `curlyQueue`:

```
imqcmd pause dst  -t q  -n curlyQueue  -pst PRODUCERS  -u admin
```
The following command pauses delivery to message consumers from the topic destination `hotTopic`:

```
imqcmd pause dst  -t t  -n hotTopic  -pst CONSUMERS  -u admin
```
This command pauses all message delivery to and from all physical destinations:

```
imqcmd pause dst  -u admin
```

> **Note:**   In a broker cluster, since each broker in the cluster has its own instance of each physical destination, you must pause each such instance individually.

The `imqcmd resume dst` subcommand resumes delivery to a paused destination:

```
imqcmd resume dest  [-t destType  -n destName]
```
For example, the following command resumes message delivery to the queue destination `curlyQueue`:

```
imqcmd resume dst  -t q  -n curlyQueue  -u admin
```
If no destination type and name are specified, all destinations are resumed. This command resumes delivery to all physical destinations:

```
imqcmd resume dst  -u admin
```

Purging a Physical Destination

Purging a physical destination deletes all messages it is currently holding. You might want to do this when a destination's accumulated messages are taking up too much of the system's resources, such as when a queue is receiving messages but has no registered consumers to which to deliver them, or when a topic's durable subscribers remain inactive for long periods of time.

To purge a physical destination, use the `imqcmd purge dst` subcommand:

```
imqcmd purge dst  -t destType  -n destName
```
For example, the following command purges all accumulated messages from the topic destination `hotTopic`:

```
imqcmd purge dst  -t t  -n hotTopic  -u admin
```

> **Note:**   In a broker cluster, since each broker in the cluster has its own instance of each physical destination, you must purge each such instance individually.

> **Tip:** When restarting a broker that has been shut down, you can use the Broker utility's `-reset messages` option to clear out its stale messages: for example,
>
> ```
> imqbrokerd -reset messages  -u admin
> ```
> This saves you the trouble of purging physical destinations after restarting the broker.

Updating Physical Destination Properties

The subcommand `imqcmd update dst` changes the values of specified properties of a physical destination:

```
imqcmd update dst  -t destType  -n destName
-o property1=value1  [ [-o property2=value2] … ]
```

The properties to be updated can include any of those listed in Table 18–1 (with the exception of the `isLocalOnly` property, which cannot be changed once the destination has been created). For example, the following command changes the `maxBytesPerMsg` property of the queue destination `curlyQueue` to `1000` and the `maxNumMsgs` property to `2000`:

```
imqcmd update dst  -t q  -n curlyQueue  -u admin
-o maxBytesPerMsg=1000
-o maxNumMsgs=2000
```

> **Note:** The *type* of a physical destination is not an updatable property; you cannot use the `imqcmd update dst` subcommand to change a queue to a topic or a topic to a queue.

Viewing Physical Destination Information

To list the physical destinations on a broker, use the `imqcmd list dst` subcommand:

```
imqcmd list dst  -b hostName:portNumber  [-t destType][-tmp]
```

This lists all physical destinations on the broker identified by *hostName* and *portNumber* of the type (queue or topic) specified by *destType*. If the `-t` option is omitted, both queues and topics are listed. For example, the following command lists all physical destinations on the broker running on host `myHost` at port number `4545`:

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

> **Note:** The list of queue destinations always includes the dead message queue (`mq.sys.dmq`) in addition to any other queue destinations currently existing on the broker.

If you specify the `-tmp` option, temporary destinations are listed as well. These are destinations created by clients, normally for the purpose of receiving replies to messages sent to other clients.

The `imqcmd query dst` subcommand displays information about a single physical destination:

```
imq query dst  -t destType  -n destName
```

For example, the following command displays information about the queue destination `curlyQueue`:

```
imqcmd query dst  -t q  -n curlyQueue  -u admin
```

Example 7–3 shows an example of the resulting output. You can use the `imqcmd update dst` subcommand (see Updating Physical Destination Properties) to change the value of any of the properties listed.

Example 7–3   Physical Destination Information Listing

```
------------------------------------
Destination Name    Destination Type
------------------------------------
large.green.circle    Topic


On the broker specified by:


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


Destination Name                    large.green.circle
Destination Type                    Topic
Destination State                   RUNNING
Created Administratively            true

Current Number of Messages
   Actual                           0
   Remote                           0
   In Delay Delivery                0
   Held in Transaction              0
Current Message Bytes
   Actual                           0
   Remote                           0
   Held in Transaction              0
Current Number of Producers         0
Current Number of Producer Wildcards   0
Current Number of Consumers         1
Current Number of Consumer Wildcards   1
   large.*.circle  (1)

Max Number of Messages              unlimited (-1)
Max Total Message Bytes             unlimited (-1)
Max Bytes per Message               unlimited (-1)
Max Number of Producers             100

Limit Behavior                      REJECT_NEWEST
Consumer Flow Limit                 1000
Is Local Destination                false
Use Dead Message Queue              true
XML schema validation enabled       false
XML schema URI List                 -
Reload XML schema on failure        false
```

For destinations in a broker cluster, it is often helpful to know how many messages in a destination are local (produced to the local broker) and how many are remote (produced to a remote broker). Hence, `imqcmd query dst` reports, in addition to the number and total message bytes of messages in the destination, the number and total bytes of messages that are sent to the destination from remote brokers in the cluster.

For topic destinations, `imqcmd query dst` reports the number of publishers that are wildcard publishers (including their corresponding symbolic destination names) and

the number of subscribers that are wildcard subscribers (including their symbolic destination names), if any.

To display metrics information about a physical destination, use the `imqcmd metrics dst` subcommand:

```
imqcmd metrics dst   -t destType  -n destName
[-m metricType]
[-int interval]
[-msp numSamples]
```

The `-m` option specifies the type of metric information to display:

- `ttl` *(default)*: Messages and packets flowing into and out of the destination and residing in memory

- `rts`: Rate of flow of messages and packets into and out of the destination per second, along with other rate information

- `con`: Metrics related to message consumers

- `dsk`: Disk usage

The `-int` and `-msp` options specify, respectively, the interval (in seconds) at which to display the metrics and the number of samples to display in the output. The default values are 5 seconds and an unlimited number of samples.

For example, the following command displays cumulative totals for messages and packets handled by the queue destination `curlyQueue`:

```
imqcmd metrics dst  -t q  -n curlyQueue  -m ttl  -u admin
```

Example 7–4 shows an example of the resulting output.

Example 7–4   Physical Destination Metrics Listing

```
-------------------------------------------------------------------------------
-
   Msgs         Msg Bytes          Msg Count        Total Msg Bytes (k)
Largest
 In   Out     In       Out    Current  Peak  Avg  Current    Peak     Avg    Msg
(k)
-------------------------------------------------------------------------------
-
3128  3066  1170102  1122340    128     409   29    46       145       10     < 1
4858  4225  1863159  1635458    144     201   33    53       181       42     < 1
2057  1763   820804   747200     84     377   16    71       122       79     < 1
```

For a more detailed description of the use of the Command utility to report physical destination metrics, see Physical Destination Metrics.

Managing Physical Destination Disk Utilization

Because of the way message storage is structured in a file-based persistent data store (see File-Based Persistence Properties), disk space can become fragmented over time, resulting in inefficient utilization of the available resources. Message Queue's Command utility (`imqcmd`) provides subcommands for monitoring disk utilization by physical destinations and for reclaiming unused disk space when utilization drops.

To monitor a physical destination's disk utilization, use the `imqcmd metrics dst` subcommand:

```
imqcmd metrics dst  -m dsk  -t destType  -n destMame
```

This displays the total number of bytes of disk space reserved for the destination's use, the number of bytes currently in use to hold active messages, and the percentage of available space in use (the disk utilization ratio). For example, the following command displays disk utilization information for the queue destination curlyQueue:

```
imqcmd metrics dst  -m dsk  -t q  -n curlyQueue  -u admin
```
Example 7–5 shows an example of the resulting output.

Example 7–5   Destination Disk Utilization Listing

```
-------------------------------------
Reserved   Used      Utilization Ratio
-------------------------------------
804096     675533    84
1793024    1636222   91
2544640    2243808   88
```

The disk utilization pattern depends on the characteristics of the messaging application using a particular physical destination. Depending on the flow of messages into and out of the destination and their relative size, the amount of disk space reserved might grow over time. If messages are produced at a higher rate than they are consumed, free records should generally be reused and the utilization ratio should be on the high side. By contrast, if the rate of message production is comparable to or lower than the consumption rate, the utilization ratio will likely be low.

As a rule, you want the reserved disk space to stabilize and the utilization ratio to remain high. If the system reaches a steady state in which the amount of reserved disk space remains more or less constant with utilization above 75%, there is generally no need to reclaim unused disk space. If the reserved space stabilizes at a utilization rate below 50%, you can use the imqcmd compact dst subcommand to reclaim the disk space occupied by free records:

```
compact dst  [-t destType  -n destName]
```
This compacts the file-based data store for the designated physical destination. If no destination type and name are specified, all physical destinations are compacted.

You must pause a destination (with the imqcmd pause subcommand) before compacting it, and resume it (with imqcmd resume) afterward (see Pausing and Resuming a Physical Destination):

```
imqcmd pause dst     -t q  -n curlyQueue  -u admin
imqcmd compact dst  -t q  -n curlyQueue  -u admin
imqcmd resume dst    -t q  -n curlyQueue  -u admin
```

> **Tip:**   If a destination's reserved disk space continues to increase over time, try reconfiguring its maxNumMsgs, maxBytesPerMsg, maxTotalMsgBytes, and limitBehavior properties (see Physical Destination Properties).

Using the Dead Message Queue

The dead message queue, mq.sys.dmq, is a system-created physical destination that holds the dead messages of a broker's physical destinations. The dead message queue is a tool for monitoring, tuning system efficiency, and troubleshooting. For a definition of the term dead message and a more detailed introduction to the dead message queue, see the *Open Message Queue Technical Overview*.

The broker automatically creates a dead message queue when it starts. The broker places messages on the queue if it cannot process them or if their time-to-live has expired. In addition, other physical destinations can use the dead message queue to hold discarded messages. This can provide information that is useful for troubleshooting the system.

Managing the Dead Message Queue

The physical destination configuration property useDMQ controls a destination's use of the dead message queue. Physical destinations are configured to use the dead message queue by default; to disable a destination from using it, set the destination's useDMQ property to false:

```
imqcmd update dst  -t q  -n curlyQueue  -o useDMQ=false
```
You can enable or disable the use of the dead message queue for all auto-created physical destinations on a broker by setting the broker's imq.autocreate.destination.useDMQ broker property:

```
imqcmd update bkr  -o imq.autocreate.destination.useDMQ=false
```
You can manage the dead message queue with the Message Queue Command utility (imqcmd) just as you manage other queues, but with some differences. For example, because the dead message queue is system-created, you cannot create, pause, or destroy it. Also, as shown in Table 7–2, default values for the dead message queue's configuration properties sometimes differ from those of ordinary queues.

Table 7–2   Dead Message Queue Treatment of Physical Destination Properties

Property	Variant Treatment by Dead Message Queue
maxNumMsgs	Default value is 1000, rather than -1 (unlimited) as for ordinary queues.
maxTotalMsgBytes	Default value is 10m (10 megabytes), rather than -1 (unlimited) as for ordinary queues.
limitBehavior	Default value is REMOVE_OLDEST, rather than REJECT_NEWEST as for ordinary queues. FLOW_CONTROL is not supported for the dead message queue.
maxNumProducers	Does not apply to the dead message queue.
isLocalOnly	Permanently set to false in broker clusters; the dead message queue in a cluster is always a global physical destination.
localDeliveryPreferred	Does not apply to the dead message queue.

> **Tip:**   By default, the dead message queue stores entire messages. If you do not plan to restore dead messages, you can reduce the size of the dead message queue by setting the broker's imq.destination.DMQ.truncateBody property to true:
>
> ```
> imqcmd update bkr  -o imq.destination.DMQ.truncateBody=true
> ```
> This will discard the body of all messages and retain only the headers and property data.

Enabling Dead Message Logging

The broker configuration property logDeadMsgs controls the logging of events related to the dead message queue. When dead message logging is enabled, the broker will log the following events:

■   A message is moved to the dead message queue.

- A message is discarded from the dead message queue (or from any physical destination that does not use the dead message queue).

- A physical destination reaches its limits.

Dead message logging is disabled by default. The following command enables it:

```
imqcmd update bkr  -o imq.destination.logDeadMsgs=true
```

> **Note:**   Dead message logging applies to all physical destinations that use the dead message queue. You cannot enable or disable logging for an individual physical destination.

Managing Broker System-Wide Memory

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

The performance and stability of a broker depend on the system resources (such as memory) available and how efficiently they are utilized. You can set configuration properties to prevent the broker from becoming overwhelmed by incoming messages or running out of memory. These properties function at three different levels to keep the message service operating as resources become scarce:

- **Systemwide message limits** apply collectively to all physical destinations on the system. These include the maximum number of messages held by a broker (`imq.system.max_count`) and the maximum total number of bytes occupied by such messages (`imq.system.max_size`). If either of these limits is reached, the broker will reject any new messages until the pending messages fall below the limit. There is also a limit on the maximum size of an individual message (`imq.message.max_size`) and a time interval at which expired messages are removed (`imq.message.expiration.interval`).

- **Individual destination limits** regulate the flow of messages to a specific physical destination. The configuration properties controlling these limits are described in Physical Destination Property Reference. They include limits on the number and size of messages the destination will hold, the number of message producers and consumers that can be created for it, and the number of messages that can be batched together for delivery to the destination.

  The destination can be configured to respond to memory limits by slowing down the delivery of message by message producers, by rejecting new incoming messages, or by throwing out the oldest or lowest-priority existing messages. Messages deleted from the destination in this way may optionally be moved to the dead message queue rather than discarded outright; the broker property `imq.destination.DMQ.truncateBody` controls whether the entire message body is saved in the dead message queue, or only the header and property data.

  As a convenience during application development and testing, you can configure a message broker to create new physical destinations automatically whenever a message producer or consumer attempts to access a nonexistent destination. The broker properties summarized in Table 17–3 parallel the ones just described, but apply to such *auto-created destinations* instead of administratively created ones.

- **System memory thresholds** define levels of memory usage at which the broker takes increasingly serious action to prevent memory overload. Four such usage levels are defined:

  - **Green:** Plenty of memory is available.

  - **Yellow:** Broker memory is beginning to run low.

  - **Orange:** The broker is low on memory.

  - **Red:** The broker is out of memory.

  The memory utilization percentages defining these levels are specified by the broker properties `imq.green.threshold`, `imq.yellow.threshold`, `imq.orange.threshold`, and `imq.red.threshold`, respectively; the default values are 0% for green, 80% for yellow, 90% for orange, and 98% for red.

  As memory usage advances from one level to the next, the broker responds progressively, first by swapping messages out of active memory into persistent storage and then by throttling back producers of nonpersistent messages, eventually stopping the flow of messages into the broker. (Both of these measures degrade broker performance.) The throttling back of message production is done by limiting the size of each batch delivered to the number of messages specified by the properties `imq.`*resourceState*`.count`, where *resourceState* is `green`, `yellow`, `orange`, or `red`, respectively.

The triggering of these system memory thresholds is a sign that systemwide and destination message limits are set too high. Because the memory thresholds cannot always catch potential memory overloads in time, you should not rely on them to control memory usage, but rather reconfigure the system-wide and destination limits to optimize memory resources.

Managing Durable Subscriptions

Message Queue clients subscribing to a topic destination can register as durable subscribers. The corresponding durable subscription has a unique, persistent identity and requires the broker to retain messages addressed to it even when its message consumer (the durable subscriber) becomes inactive. Ordinarily, the broker may delete a message held for a durable subscriber only when the message expires.

The Message Queue Command utility provides subcommands for managing a broker's durable subscriptions in the following ways:

- Listing durable subscriptions

- Purging all messages for a durable subscription

- Destroying a durable subscription

To list durable subscriptions for a specified physical destination, use the `imqcmd list dur` subcommand:

```
imqcmd list dur  -d topicName
```
For example, the following command lists all durable subscriptions to the topic `SPQuotes` on the default broker (host `localhost` at port `7676`):

```
imqcmd list dur  -d SPQuotes
```
The resulting output lists the name of each durable subscription to the topic, the client identifier to which it belongs, its current state (active or inactive), and the number of messages currently queued to it. Example 7–6 shows an example.

Example 7–6   Durable Subscription Information Listing

```
Name        Client ID       Number of   Durable Sub
                            Messages      State
----------------------------------------------------------------
myDurable   myClientID       1            INACTIVE
```

The `imqcmd purge dur` subcommand purges all messages for a specified durable
subscriber and client identifier:

```
imqcmd purge dur  -n subscriberName
                     -c clientID
```

For example, the following command purges all messages for the durable subscription
listed in :

```
imqcmd purge dur  -n myCurable  -c myClientID
```

The `imqcmd destroy dur` subcommand destroys a durable subscription, specified by its
subscriber name and client identifier:

```
imqcmd destroy dur  -n subscriberName
                       -c clientID
```

For example, the following command destroys the durable subscription listed in
:

```
imqcmd destroy dur  -n myCurable  -c myClientID
```

Managing Transactions

All transactions initiated by client applications are tracked by the broker. These can be
local Message Queue transactions or distributed transactions managed by a
distributed transaction manager.

Each transaction is identified by a unique 64-bit Message Queue transaction identifier.
Distributed transactions also have a distributed transaction identifier (XID), up to 128
bytes long, assigned by the distributed transaction manager. Message Queue
maintains the association between its own transaction identifiers and the
corresponding XIDs.

The `imqcmd list txn` subcommand lists the transactions being tracked by a broker:

```
imqcmd list txn
```

This lists all transactions on the broker, both local and distributed. For each
transaction, it shows the transaction ID, state, user name, number of messages and
acknowledgments, and creation time. shows an example of the resulting
output.

Example 7–7   Broker Transactions Listing

```
----------------------------------------------------------------
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
```

To display detailed information about a single transaction, obtain the transaction
identifier from `imqcmd list txn` and pass it to the `imqcmd query txn` subcommand:

```
imqcmd query txn  -n transactionID
```
This displays the same information as `imqcmd list txn`, along with the client identifier, connection identification, and distributed transaction identifier (XID). For example, the command

```
imqcmd query txn  -n 64248349708800
```
produces output like that shown in Example 7–8.

Example 7–8   Transaction Information Listing

```
Client ID
Connection              guest@192.18.116.219:62209->jms:62195
Creation time           1/30/02 10:08:31 AM
Number of acknowledgments  0
Number of messages      4
State                   PREPARED
Transaction ID          64248349708800
User name               guest
XID
6469706F6C7369646577696E646572313031323343134313130030373230
```

If a broker fails, it is possible that a distributed transaction could be left in the `PREPARED` state without ever having been committed. Until such a transaction is committed, its messages will not be delivered and its acknowledgments will not be processed. Hence, as an administrator, you might need to monitor such transactions and commit them or roll them back manually. For example, if the broker's `imq.transaction.autorollback` property (see Table 17–2) is set to `false`, you must manually commit or roll back non-distributed transactions and unrecoverable distributed transactions found in the `PREPARED` state at broker startup, using the Command utility's `commit txn` or `rollback txn` subcommand:

```
imqcmd commit txn  -n transactionID
imqcmd rollback txn  -n transactionID
```
For example, the following command commits the transaction listed in Example 7–8:

```
imqcmd commit txn  -n64248349708800
```

> **Note:**   Only transactions in the PREPARED state can be committed. However, transaction in the `STARTED`, `FAILED`, `INCOMPLETE`, `COMPLETE`, and `PREPARED` states can be rolled back. You should do so only 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.

8

Configuring Persistence Services

For a broker to recover in case of failure, it needs to re-create the state of its message delivery operations. To do this, the broker must save state information to a *persistent data store.* When the broker restarts, it uses the saved data to re-create destinations and durable subscriptions, recover persistent messages, roll back open transactions, and rebuild its routing table for undelivered messages. It can then resume message delivery.
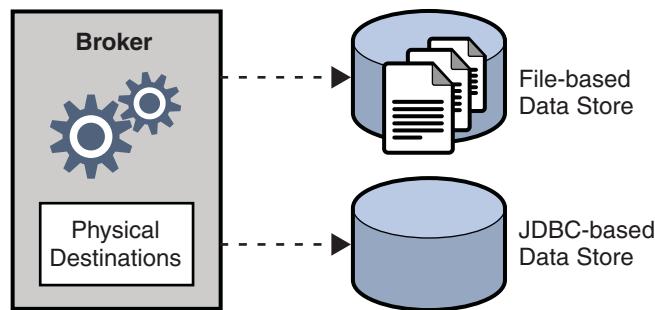
A persistent data store is thus a key aspect of providing for reliable message delivery. This chapter describes the two different persistence implementations supported by the Message Queue broker and how to set each of them up:

- Introduction to Persistence Services
- File-Based Persistence
- JDBC-Based Persistence
- Data Store Formats

Introduction to Persistence Services

A broker's persistent data store holds information about physical destinations, durable subscriptions, messages, transactions, and acknowledgments.

Message Queue supports both file-based and JDBC-based persistence modules, as shown in the following figure. File-based persistence uses individual files to store persistent data; JDBC-based persistence uses the Java Database Connectivity (JDBC) interface to connect the broker to a JDBC-based data store. While file-based persistence is generally faster than JDBC-based persistence, some users prefer the redundancy and administrative control provided by a JDBC database. The broker configuration property `imq.persist.store` (see Table 17–5) specifies which of the two persistence modules (`file` or `jdbc`) to use.

Figure 8–1   Persistent Data Stores

Message Queue brokers are configured by default to use a file-based persistent store, but you can reconfigure them to plug in any data store accessible through a JDBC-compliant driver. The broker configuration property `imq.persist.store` (see Table 17–5) specifies which of the two forms of persistence to use.

File-Based Persistence

By default, Message Queue uses a file-based data store, in which individual files store persistent data (such as messages, destinations, durable subscriptions, transactions, and routing information).

The file-based data store is located in a directory identified by the name of the broker instance (*instanceName*) to which the data store belongs:

```
IMQ_VARHOME/instances/instanceName/fs370
```
Each destination on the broker has its own subdirectory holding messages delivered to that destination.

> **Note:**   Because the data store can contain messages of a sensitive or proprietary nature, you should secure the `IMQ_VARHOME/instances/instanceName/fs370` directory against unauthorized access; see Securing a File-Based Data Store.

File-Based Persistence Properties

Broker configuration properties related to file-based persistence are listed under File-Based Persistence Properties. These properties let you configure various aspects of how the file-based data store behaves.

All persistent data other than messages is stored in separate files: one file each for destinations, durable subscriptions, and transaction state information. Most messages are stored in a single file consisting of variable-size records. You can compact this file to alleviate fragmentation as messages are added and removed (see Managing Physical Destination Disk Utilization). In addition, messages above a certain threshold size are stored in their own individual files rather than in the variable-sized record file. You can configure this threshold size with the broker property `imq.persist.file.message.max_record_size`.

The broker maintains a file pool for these individual message files: instead of being deleted when it is no longer needed, a file is returned to the pool of free files in its destination directory so that it can later be reused for another message. The broker property `imq.persist.file.destination.message.filepool.limit` specifies the maximum number of files in the pool. When the number of individual message files

for a destination exceeds this limit, files will be deleted when no longer needed instead of being returned to the pool.

When returning a file to the file pool, the broker can save time at the expense of storage space by simply tagging the file as available for reuse without deleting its previous contents. You can use the `imq.persist.file.message.filepool.cleanratio` broker property to specify the percentage of files in each destination's file pool that should be maintained in a "clean" (empty) state rather than simply marked for reuse. The higher you set this value, the less space will be required for the file pool, but the more overhead will be needed to empty the contents of files when they are returned to the pool. If the broker's `imq.persist.file.message.cleanup` property is `true`, all files in the pool will be emptied at broker shutdown, leaving them in a clean state; this conserves storage space but slows down the shutdown process.

In writing data to the data store, the operating system has some leeway in whether to write the data synchronously or "lazily" (asynchronously). Lazy storage can lead to data loss in the event of a system crash, if the broker believes the data to have been written to the data store when it has not. To ensure absolute reliability (at the expense of performance), you can require that all data be written synchronously by setting the broker property `imq.persist.file.sync.enabled` to `true`. In this case, the data is guaranteed to be available when the system comes back up after a crash, and the broker can reliably resume operation.

Configuring a File-Based Data Store

A file-based data store is automatically created when you create a broker instance. However, you can configure the data store using the properties described in File-Based Persistence Properties.

For example, by default, Message Queue performs asynchronous write operations to disk. However, to attain the highest reliability, you can set the broker property `imq.persist.file.sync` to write data synchronously instead. See Table 17–6.

When you start a broker instance, you can use the `imqbrokerd` command's `-- reset` option to clear the file-based data store. For more information about this option and its suboptions, see Broker Utility.

Securing a File-Based Data Store

The persistent data store can contain, among other information, message files that are being temporarily stored. Since these messages may contain proprietary information, it is important to secure the data store against unauthorized access. This section describes how to secure data in a file-based data store.

A broker using file-based persistence writes persistent data to a flat-file data store:

`IMQ_VARHOME/instances/`*instanceName*`/fs370`
where *instanceName* is a name identifying the broker instance. This directory is created when the broker instance is started for the first time. The procedure for securing this directory depends on the operating system platform on which the broker is running:

- On Solaris and Linux, the directory's permissions are determined by the file mode creation mask (`umask`) of the user who started the broker instance. Hence, permission to start a broker instance and to read its persistent files can be restricted by setting the mask appropriately. Alternatively, an administrator (superuser) can secure persistent data by setting the permissions on the `instances` directory to `700`.

- On Windows, the directory's permissions can be set using the mechanisms provided by the Windows operating system. This generally involves opening a Properties dialog for the directory.

Optimizing File-Based Transaction Persistence

Because many activities can occur during a transaction, persisting a transaction's state over the complete life cycle of the transaction can adversely affect overall performance, especially when the `imq.persist.file.sync.enabled` property is set to `true` to avoid data loss in case of a system crash.

Message Queue provides a transaction logging mechanism that can improve performance of transaction persistence. This transaction log offers tuning parameters that can improve performance of file-based persistence for other objects, such as message payloads.

To enable this transaction logging mechanism, set the `imq.persist.file.newTxnLogenabled` broker property to `true`.

After enabling the transaction log, essential changes to the state of a JMS transaction are written to the transaction log. When the transaction is committed, all details regarding it are gathered and written to the persistent store. Additionally, the logging mechanism periodically performs a "checkpoint" operation to ensure that the persistent store and the transaction log are synchronized and that the log size remains manageable.

As a further refinement, the operation of the logging mechanism is subject to the value of the `imq.persist.file.sync.enabled` broker property:

- When `imq.persist.file.sync.enabled` is `true`, write operations to the transaction log are written synchronously to disk. Non-transacted message and non-transacted message acknowledgements are also written synchronously to the transaction log before being written asynchronously to the persistent store.

- When `imq.persist.file.sync.enabled` is `false`, write operations to the transaction log are written asynchronously to disk. Non-transacted message and non-transacted message acknowledgements are not written to the transaction log.

The tuning parameters supported by the transaction logging mechanism are:

- `imq.persist.file.txnLog.groupCommit`

- `imq.persist.file.txnLog.logNonTransactedMsgSend`

- `imq.persist.file.txnLog.logNonTransactedMsgAck`

Information about these parameters can be found in Table 17–7.

JDBC-Based Persistence

Instead of using a file-based data store, you can set up a broker to access any data store accessible through a JDBC-compliant driver. This involves setting the appropriate JDBC-related broker configuration properties and using the Database Manager utility (`imqdbmgr`) to create the proper database schema. See Configuring a JDBC-Based Data Store for specifics.

JDBC-Based Persistence Properties

The full set of properties for configuring a broker to use a JDBC database are listed in Table 17–8. You can specify these properties either in the instance configuration file

(`config.properties`) of each broker instance or by using the `-D` command line option to the Broker utility (`imqbrokerd`) or the Database Manager utility (`imqdbmgr`).

In practice, however, JDBC properties are preconfigured by default, depending on the database vendor being used for the data store. The property values are set in the `default.properties` file, and only need to be explicitly set if you are overriding the default values. In general, you only need to set the following properties:

- `imq.persist.store`

  This property specifies that a JDBC-based data store (as opposed to the default file-based data store) is used to store persistent data.

- `imq.persist.jdbc.dbVendor`

  This property identifies the database vendor being used for the data store; all of the remaining properties are qualified by this vendor name.

- `imq.persist.jdbc`*vendorName*`.user`

  This property specifies the user name to be used by the broker in accessing the database.

- `imq.persist.jdbc`*vendorName*`.password`

  This property specifies the password for accessing the database, if required; `imq.persist.jdbc.`*vendorName*`.needpassword` is a boolean flag specifying whether a password is needed. For security reasons, the database access password should be specified only in a password file referenced with the `-passfile` command line option; if no such password file is specified, the `imqbrokerd` and `imqdbmgr` commands will prompt for the password interactively.

- `imq.persist.jdbc.`*vendorName*`.property.`*propName*

  This set of properties represents any additional, vendor-specific properties that are required.

- `imq.persist.jdbc.`*vendorName*`.tableoption`

  Specifies the vendor-specific options passed to the database when creating the table schema.

Example 8–1   Broker Properties for MySQL Database

```
imq.persist.store=jdbc
imq.persist.jdbc.dbVendor=mysql
imq.persist.jdbc.mysql.user=userName
imq.persist.jdbc.mysql.password=password
imq.persist.jdbc.mysql.property.url=jdbc:mysql://hostName:port/dataBase
```

If you expect to have messages that are larger than 1 MB, configure MySQL's `max_allowed_packet` variable accordingly when starting the database. For more information see Appendix B of the *MySQL 5.0 Reference Manual*.

In addition, in an enhanced broker cluster, in which a JDBC database is shared by multiple broker instances, each broker must be uniquely identified in the database (unnecessary for an embedded database, which stores data for only one broker instance). The configuration property `imq.brokerid` specifies a unique instance identifier to be appended to the names of database tables for each broker. See Enhanced Broker Cluster Properties.

After setting all of the broker's needed JDBC configuration properties, you must also install your JDBC driver's `.jar` file in `IMQ_HOME/lib/ext` and then create the database schema for the JDBC-based data store (see To Set Up a JDBC-Based Data Store).

Configuring a JDBC-Based Data Store

To configure a broker to use a JDBC database, you set JDBC-related properties in the broker's instance configuration file and create the appropriate database schema. The Message Queue Database Manager utility (`imqdbmgr`) uses your JDBC driver and the broker configuration properties to create the schema and manage the database. You can also use the Database Manager to delete corrupted tables from the database or if you want to use a different database as a data store. See Database Manager Utility for more information.

Note: If you use an embedded database, it is best to create it under the following directory:

`.../instances/`*instanceName*`/dbstore/`*databaseName*

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.

To Set Up a JDBC-Based Data Store

1. Set JDBC-related properties in the broker's instance configuration file.

   The relevant properties are discussed, with examples, in JDBC-Based Persistence Properties and listed in full in Table 17–8. In particular, you must specify a JDBC-based data store by setting the broker's `imq.persist.store` property to `jdbc`.

2. Place a copy of, or a symbolic link to, your JDBC driver's `.jar` file in `IMQ_HOME/lib/ext`, the Message Queue external resource files directory.

3. Check the message table schema for your database in `IMQ_HOME/lib/props/broker/default.properties`.

   In the `default.properties` file, locate the message table schema for your database by searching for "imq.persist.jdbc.*vendor*.table.MQMSG". Read any commentary notes about the schema and check that the maximum message size is sufficient; if it is not, adjust the size of the MESSAGE column as needed.

4. Create the database schema needed for Message Queue 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.

   a. Change to the directory where the Database Manager utility resides:

      `cd IMQ_HOME/bin`

   b. Enter the `imqdbmgr` command:

      `imqdbmgr create all`

5. Optionally, if your JDBC driver does not provide automatic reconnect, you can configure the following:

   - `imq.persist.jdbc.connection.retry.delay` specifies the amount of time, in milliseconds, the broker waits before it attempts to retry connecting to a database server for a failed connection.

- `imq.persist.jdbc.connection.retry.max` specifies the maximum number of times the broker will retry connecting to a database server after a failed connection.

- `imq.persist.jdbc.`*`vendorName`*`.connection.retry.regex.`*`n`* specifies 1 or more regular expressions, to be used to match `SQLException` messages in order to identify retriable database communication failures.

To Display Information About a JDBC-Based Data Store

You can display information about a JDBC-based data store using the Database Manager utility (`imqdbmgr`) as follows:

1. Change to the directory where the Database Manager utility resides:

```
cd IMQ_HOME/bin
```

2. Enter the `imqdbmgr` command:

```
imqdbmgr query
```

The output should resemble the following

```
dbmgr query

[04/Oct/2005:15:30:20 PDT] Using plugged-in persistent store:
        version=400
        brokerid=Mozart1756
        database connection url=jdbc:oracle:thin:@Xhome:1521:mqdb
        database user=scott
Running in standalone mode.
Database tables have already been created.
```

Securing a JDBC-Based Data Store

The persistent data store can contain, among other information, message files that are being temporarily stored. Since these messages may contain proprietary information, it is important to secure the data store against unauthorized access. This section describes how to secure data in a JDBC-based data store.

A broker using JDBC-based persistence writes persistent data to a JDBC-compliant database. For a database managed by a database server (such as Oracle), it is recommended that you create a user name and password to access the Message Queue database tables (tables whose names start with `MQ`). If the database does not allow individual tables to be protected, create a dedicated database to be used only by Message Queue brokers. See the documentation provided by your database vendor for information on how to create user name/password access.

The user name and password required to open a database connection by a broker can be provided as broker configuration properties. However it is more secure to provide them as command line options when starting up the broker, using the `imqbrokerd` command's `-dbuser`and `-dbpassword` options (see Broker Utility).

For an embedded database that is accessed directly by the broker by means of the database's JDBC driver, security is usually provided by setting file permissions on the directory where the persistent data will be stored, as described above under Securing a File-Based Data Store To ensure that the database is readable and writable by both the broker and the Database Manager utility, however, both should be run by the same user.

Data Store Formats

Changes in the file formats for both file-based and JDBC-based persistent data stores were introduced in Message Queue 3.7, with further JDBC changes in version 4.0 and 4.1. As a result of these changes, the persistent data store version numbers have been updated to 370 for file-based data stores and 410 for JDBC-based stores. You can use the `imqdbmgr query` command to determine the version number of your existing data store.

On first startup, the Message Queue Broker utility (`imqbrokerd`) will check for the presence of an older persistent data store and automatically migrate it to the latest format:

- File-based data store versions 200 and 350 are migrated to the version 370 format.

- JDBC-based data store versions 350, 370, and 400 are migrated to the version 410 format. (If you need to upgrade a version 200 data store, you will need to step through an intermediate Message Queue 3.5 or 3.6 release.)

The upgrade leaves the older copy of the persistent data store intact, allowing you to roll back the upgrade if necessary. To do so, you can uninstall the current version of Message Queue and reinstall the earlier version you were previously running. The older version's message brokers will locate and use the older copy of the data store.

Beginning in Message Queue 4.5, the `imq.persist.file.newTxnLog` property is `true` by default. This setting can generate an error when starting a broker that is using an older persistent data store. To resolve the error, set the property value to `false` and start the broker, thus migrating the data store to the latest format. Then, you can stop the broker, set the property value back to `true`, and start the broker without encountering an error.

9

Configuring and Managing Security Services

This chapter describes Message Queue's facilities for security-related administration tasks, such as configuring user authentication, defining access control, configuring a Secure Socket Layer (SSL) connection service to encrypt client-broker communication, and setting up a password file for administrator account passwords. In addition to Message Queue's own built-in authentication mechanisms, you can also plug in a preferred external service based on the Java Authentication and Authorization Service (JAAS) API.

This chapter includes the following sections:

- Introduction to Security Services
- User Authentication
- User Authorization
- Message Encryption
- Password Files
- Connecting Through a Firewall
- Audit Logging with the Solaris BSM Audit Log

Introduction to Security Services
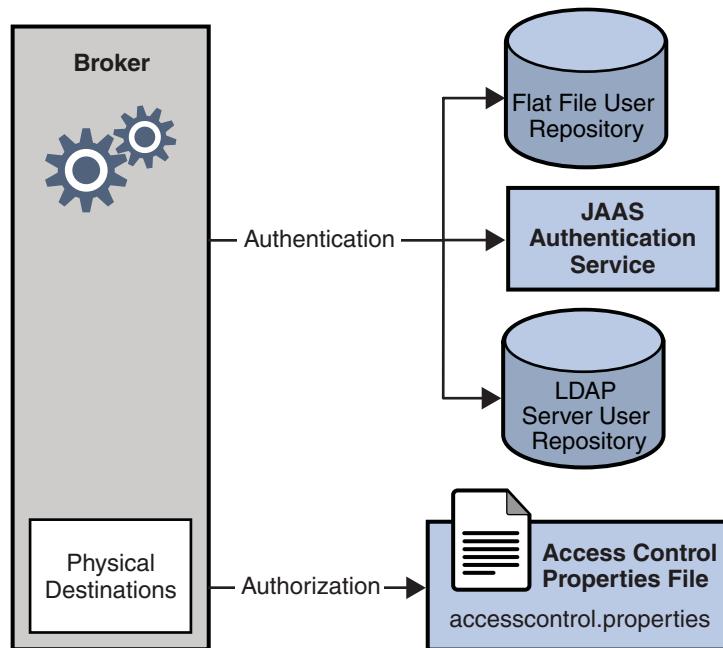
Message Queue provides security services for user access control (authentication and authorization) and for encryption:

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

As a Message Queue administrator, you are responsible for setting up the information the broker needs to authenticate users and authorize their actions. The broker properties pertaining to security services are listed under Security Properties. The boolean property `imq.accesscontrol.enabled` acts as a master switch that controls whether access control is applied on a brokerwide basis; for finer control, you can override this setting for a particular connection service by setting the `imq.`*serviceName*`.accesscontrol.enabled` property, where *serviceName* is the name of the connection service, as shown in Table 6–1: for example, `imq.httpjms.accesscontrol.enabled`.

The following figure shows the components used by the broker to provide authentication and authorization services. These services depend on a *user repository* containing information about the users of the messaging system: their names, passwords, and group memberships. In addition, to authorize specific operations for a user or group, the broker consults an *access control file* that specifies which operations a user or group can perform. You can designate a single access control file for the broker as a whole, using the configuration property imq.accesscontrol.file.filename, or for a single connection service with imq.*serviceName*. accesscontrol.file.filename.

Figure 9–1   Security Support

As Figure 9–1 shows, you can store user data in a flat file user repository that is provided with the Message Queue service, you can access an existing LDAP repository, or you can plug in a Java Authentication and Authorization Service (JAAS) module.

- If you choose a flat-file repository, you must use the imqusermgr utility to manage the repository. This option is easy to use and built-in.

- If you want to use an existing LDAP server, you use the tools provided by the LDAP vendor to populate and manage the user repository. You must also set properties in the broker instance configuration file to enable the broker to query the LDAP server for information about users and groups.

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

- If you want to plug-in an existing JAAS authentication service, you need to set the corresponding properties in the broker instance configuration file.

The broker's imq.authentication.basic.user_repository property specifies which type of repository to use. In general, an LDAP repository or JAAS authentication service is preferable if scalability is important or if you need the repository to be shared by different brokers (if you are using broker clusters, for instance). See User Authentication for more information on setting up a flat-file user repository, LDAP access, or JAAS authentication service.

Authentication

A client requesting a connection to a broker must supply a user name and password, which the broker compares with those stored in the user repository. Passwords transmitted from client to broker are encoded using either base-64 encoding (for flat-file repositories) or message digest (MD5) hashing (for LDAP repositories). The choice is controlled by the `imq.authentication.type` property for the broker as a whole, or by `imq.`*serviceName*`.authentication.type` for a specific connection service. The `imq.authentication.client.response.timeout` property sets a timeout interval for authentication requests.

As described under Password Files, you can choose to put your passwords in a *password file* instead of being prompted for them interactively. The boolean broker property `imq.passfile.enabled` controls this option. If this property is true, the `imq.passfile.dirpath` and `imq.passfile.name` properties give the directory path and file name for the password file. The `imq.imqcmd.password` property (which can be embedded in the password file) specifies the password for authenticating an administrative user to use the Command utility (`imqcmd`) for managing brokers, connection services, connections, physical destinations, durable subscriptions, and transactions.

If you are using an LDAP-based user repository, there are a whole range of broker properties available for configuring various aspects of the LDAP lookup. The address (host name and port number) of the LDAP server itself is specified by `imq.user_repository.ldap.server`. The `imq.user_repository.ldap.principal` property gives the distinguished name for binding to the LDAP repository, while `imq.user_repository.ldap.password` supplies the associated password. Other properties specify the directory bases and optional JNDI filters for individual user and group searches, the provider-specific attribute identifiers for user and group names, and so forth; see Security Properties for details.

Authorization

Once authenticated, a user can be authorized to perform various Message Queue-related activities. As a Message Queue administrator, you can define user groups and assign individual users membership in them. The default access control file explicitly refers to only one group, `admin` (see User Groups and Status). A user in this group has connection permission for the `admin` connection service, which allows the user to perform administrative functions such as creating destinations and monitoring and controlling a broker. A user in any other group that you define cannot, by default, get an `admin` service connection.

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

- Connecting to a broker

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

- Auto-creating destinations

For information on configuring authorization, see User Authorization.

Encryption

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

To use an SSL-based Message Queue connection service, you generate a public/private key pair using the Message Queue Key Tool utility (`imqkeytool`). This utility embeds the public key in a self-signed certificate and places it in a Message Queue key store. The key store is itself password-protected; to unlock it, you must provide a key store password at startup time, specified by the `imq.keystore.password` property. Once the key store 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.

For information on configuring encryption, see Message Encryption.

User Authentication

Users attempting to connect to a Message Queue message broker must provide a user name and password for authentication. The broker will grant the connection only if the name and password match those in a broker-specific user repository listing the authorized users and their passwords. Each broker instance can have its own user repository, which you as an administrator are responsible for maintaining. This section tells how to create, populate, and manage the user repository.

Message Queue can support any of three types of authentication mechanism:

- A **flat-file repository** that is shipped with Message Queue. This type of repository is very easy to populate and manage, using the Message Queue User Manager utility (`imqusermgr`). See Using a Flat-File User Repository.

- A **Lightweight Directory Access Protocol (LDAP) server.** This could be a new or existing LDAP directory server using the LDAP v2 or v3 protocol. You use the tools provided by the LDAP vendor to populate and manage the user repository. This type of repository is not as easy to use as the flat-file repository, but it is more scalable and therefore better for production environments. See Using an LDAP User Repository.

- An external authentication mechanism plugged into Message Queue by means of the Java Authentication and Authorization Service (JAAS) API. See Using JAAS-Based Authentication.

Using a Flat-File User Repository

Message Queue provides a built-in flat-file user repository and a command line tool, the User Manager utility (`imqusermgr`), for populating and managing it. Each broker has its own flat-file user repository, created automatically when you start the broker. By default, the user repository resides in a file named `passwd`, in a directory identified by the name of the broker instance with which the repository is associated:

```
IMQ_VARHOME/instances/instanceName/etc/passwd
```
If you have changed these file name or directory defaults using the `imq.user_repository.file.filename` or `imq.user_repository.file.dirpath` broker properties, you must use the `-D` option to specify the non-default values when running `imqusermgr` if you did not specify the values in the broker properties file. For example, if `imq.user_repository.file.dirpath` is specified in a cluster configuration file, start `imqusermgr` using the form:

```
imqusermgr -Dimq.cluster.url=location-of-cluster-properties-file ...
```

As another example, if `imq.user_repository.file.filename` is specified on the `imqbrokerd` command line to start the broker, start `imqusermgr` using the form:

```
imqusermgr -Dimq.user_repository.file.filename=filename-used-in-imqbrokerd-command
...
```

User Groups and Status

Each user in the repository can be assigned to a user group, which defines the default access privileges granted to all of its members. You can then specify authorization rules to further restrict these access privileges for specific users, as described in User Authorization. A user's group is assigned when the user entry is first created, and cannot be changed thereafter. The only way to reassign a user to a different group is to delete the original user entry and add another entry specifying the new group.

The flat-file user repository provides three predefined groups:

admin
For broker administrators. By default, users in this group are granted the access privileges needed to configure, administer, and manage message brokers.

user
For normal (non-administrative) client users. Newly created user entries are assigned to this group unless otherwise specified. By default, users in this group can connect to all Message Queue connection services of type NORMAL, produce messages to or consume messages from all physical destinations, and browse messages in any queue.

anonymous
For Message Queue clients that do not wish to use a user name known to the broker (for instance, because they do not know of a real user name to use). This group is analogous to the anonymous account provided by most FTPservers. No more than one user at a time can be assigned to this group. You should restrict the access privileges of this group in comparison to the user group, or remove users from the group at deployment time.

You cannot rename or delete these predefined groups or create new ones.

In addition to its group, each user entry in the repository has a user status: either active or inactive . New user entries added to the repository are marked active by default. Changing a user's status to inactive rescinds all of that user's access privileges, making the user unable to open new broker connections. Such inactive entries are retained in the user repository, however, and can be reactivated at a later time. If you attempt to add a new user with the same name as an inactive user already in the repository, the operation will fail; you must either delete the inactive user entry or give the new user a different name.

To allow the broker to be used immediately after installation without further intervention by the administrator, the flat-file user repository is created with two initial entries, summarized in Table 9–1:

- The `admin` entry (user name and password `admin/admin`) enables you to administer the broker with Command utility (`imqcmd`) commands. Immediately on installation, you should update this initial entry to change its password (see Changing a User's Password).

- The `guest` entry allows clients to connect to the broker using a default user name and password (`guest/guest`).

You can then proceed to add any additional user entries you need for individual users of your message service.

Table 9–1 Initial Entries in Flat-File User Repository

User Name	Password	Group	Status
admin	admin	admin	Active
guest	guest	anonymous	Active

Using the User Manager Utility

The Message Queue User Manager utility (`imqusermgr`) enables you to populate or edit a flat-file user repository. See User Manager Utility for general reference information about the syntax, subcommands, and options of the `imqusermgr` command.

User Manager Preliminaries Before using the User Manager, keep the following things in mind:

■ The `imqusermgr` command must be run on the host where the broker is installed.

■ If a broker-specific user repository does not yet exist, you must start up the corresponding broker instance to create it.

■ You must have appropriate permissions to write to the repository; in particular, on Solaris and Linux platforms, you must be logged in as the root user or the user who first created the broker instance.

Subcommands and General Options Table 9–2 lists the subcommands of the `imqusermgr` command. For full reference information about these subcommands, see Table 16–15.

Table 9–2 User Manager Subcommands

Subcommand	Description
add	Add user and password to repository
delete	Delete user from repository
update	Set user's password or active status (or both)
list	Display user information

The general options listed in Table 9–3 apply to all subcommands of the `imqusermgr` command.

Table 9–3 General User Manager Options

Option	Description
-D *brokerProperty=value*	Specify a broker property value when starting `imqusermgr`.
-f	Perform action without user confirmation
-s	Silent mode (no output displayed)
-v	Display version information[1]
-h	Display usage help[1]

[1] Any other options specified on the command line are ignored.

Displaying the Product Version To display the Message Queue product version, use the `-v` option. For example:

```
imqusermgr  -v
```
If you enter an `imqusermgr` command line containing the `-v` option in addition to a subcommand or other options, the User Manager utility processes only the `-v` option. All other items on the command line are ignored.

Displaying Help To display help on the `imqusermgr` command, use the `-h` option, and do not use a subcommand. You cannot get help about specific subcommands.

For example, the following command displays help about `imqusermgr`:

```
imqusermgr  -h
```
If you enter an `imqusermgr` command line containing the `-h` option in addition to a subcommand or other options, the Command utility processes only the `-h` option. All other items on the command line are ignored.

Adding a User to the Repository The subcommand `imqusermgr add` adds an entry to the user repository, consisting of a user name and password:

```
imqusermgr add  [-i brokerName]
-u userName  -p password
                [-g group]
```
The `-u` and `-p` options specify the user name and password, respectively, for the new entry. These must conform to the following conventions:

- All user names and passwords must be at least one character long. Their maximum length is limited only by command shell restrictions on the maximum number of characters that can be entered on a command line.

- A user name cannot contain an asterisk (*), a comma (,), a colon (:), or a new-line or carriage-return character.

- If a user name or password contains a space, the entire name or password must be enclosed in quotation marks (" ").

The optional `-g` option specifies the group (`admin`, `user`, or `anonymous`) to which the new user belongs; if no group is specified, the user is assigned to the `user` group by default. If the broker name (`-i` option) is omitted, the default broker `imqbroker` is assumed.

For example, the following command creates a user entry on broker `imqbroker` for a user named `AliBaba`, with password `Sesame`, in the `admin` group:

```
imqusermgr add  -u AliBaba  -p Sesame  -g admin
```

Deleting a User From the Repository The subcommand `imqusermgr delete` deletes a user entry from the repository:

```
imqusermgr delete  [-i brokerName]
-u userName
```
The `-u` option specifies the user name of the entry to be deleted. If the broker name (`-i` option) is omitted, the default broker `imqbroker` is assumed.

For example, the following command deletes the user named `AliBaba` from the user repository on broker `imqbroker`:

```
imqusermgr delete  -u AliBaba
```

Changing a User's Password You can use the subcommand `imqusermgr update` to change a user's password:

```
imqusermgr update  [-i brokerName]
-u userName  -p password
```
The -u identifies the user; -p specifies the new password. If the broker name (-i option) is omitted, the default broker imqbroker is assumed.

For example, the following command changes the password for user AliBaba to Shazam on broker imqbroker:

```
imqusermgr update  -u AliBaba  -p Shazam
```

> **Note:** For the sake of security, you should change the password of the admin user from its initial default value (admin) to one that is known only to you. The following command changes the default administrator password for broker mybroker to veeblefetzer:
>
> ```
> imqusermgr update  -i mybroker  -u admin  -p veeblefetzer
> ```
>
> 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 will prompt you for a password:
>
> ```
> imqcmd list svc  mybroker  -u admin
> ```
>
> Entering the new password (veeblefetzer) should work; the old password should fail.
>
> After changing the password, you should supply the new password whenever you use any of the Message Queue administration tools, including the Administration Console.

Activating or Deactivating a User The imqusermgr update subcommand can also be used to change a user's active status:

```
imqusermgr update  [-i brokerName]
-u userName  -a activeStatus
```
The -u identifies the user; -a is a boolean value specifying the user's new status as active (true) or inactive (false). If the broker name (-i option) is omitted, the default broker imqbroker is assumed.

For example, the following command sets user AliBaba's status to inactive on broker imqbroker:

```
imqusermgr update  -u AliBaba  -a false
```
This renders AliBabe unable to open new broker connections.

You can combine the -p (password) and -a (active status) options in the same imqusermgr update command. The options may appear in either order: for example, both of the following commands activate the user entry for AliBaba and set the password to plugh:

```
imqusermgr update  -u AliBaba  -p plugh  -a true
imqusermgr update  -u AliBaba  -a true  -p plugh
```

Viewing User Information The imqusermgr list command displays information about a user in the user repository:

```
imqusermgr list  [-i brokerName]
[-u userName]
```
The command

```
imqusermgr list  -u AliBaba
```
displays information about user AliBabe, as shown in Example 9–1.

Example 9–1   Viewing Information for a Single User

```
User repository for broker instance: imqbroker
---------------------------------
User Name    Group    Active State
---------------------------------
AliBaba      admin    true
```

If you omit the -u option

```
imqusermgr list
```
the command lists information about all users in the repository, as in Example 9–2.

Example 9–2   Viewing Information for All Users

```
User repository for broker instance: imqbroker
-------------------------------------
User Name     Group        Active State
-------------------------------------
admin         admin        true
guest         anonymous    true
AliBaba       admin        true
testuser1     user         true
testuser2     user         true
testuser3     user         true
testuser4     user         false
testuser5     user         false
```

Using an LDAP User Repository

You configure a broker to use an LDAP directory server by setting the values for certain configuration properties in the broker's instance configuration file (config.properties). These properties enable the broker instance to query the LDAP server for information about users and groups when a user attempts to connect to the broker or perform messaging operations.

- The imq.authentication.basic.user_repository property specifies the kind of user authentication the broker is to use. By default, this property is set to file, for a flat-file user repository. For LDAP authentication, set it to ldap instead:

  ```
  imq.authentication.basic.user_repository =ldap
  ```
- The imq.authentication.type property controls the type of encoding used when passing a password between client and broker. By default, this property is set to digest, denoting MD5 encoding, the form used by flat-file user repositories. For LDAP authentication, set it to basic instead:

  ```
  imq.authentication.type=basic
  ```
  This denotes base-64 encoding, the form used by LDAP user repositories.

- The following properties control various aspects of LDAP access. See Table 17–11 for more detailed information:

  ```
  imq.user_repository.ldap.server
  imq.user_repository.ldap.principal
  imq.user_repository.ldap.password
  imq.user_repository.ldap.propertyName
  imq.user_repository.ldap.base
  ```

```
imq.user_repository.ldap.uidattr
imq.user_repository.ldap.usrfilter
imq.user_repository.ldap.grpsearch
imq.user_repository.ldap.grpbase
imq.user_repository.ldap.gidattr
imq.user_repository.ldap.memattr
imq.user_repository.ldap.grpfilter
imq.user_repository.ldap.timeout
imq.user_repository.ldap.ssl.enabled
```

- ■ The `imq.user_repository.ldap.userformat` property, if set to a value of `dn`, specifies that the login username for authentication be in DN username format (for example: `uid=mquser,ou=People,dc=red,dc=sun,dc=com`). In this case, the broker extracts the value of the `imq.user.repository.lpdap.uidatr` attribute from the DN username, and uses this value as the user name in access control operations (see User Authorization).

- ■ If you want the broker to use a secure, encrypted SSL (Secure Socket Layer) connection for communicating with the LDAP server, set the broker's `imq.user_repository.ldap.ssl.enabled` property to `true`

  ```
  imq.user_repository.ldap.ssl.enabled =true
  ```
  and the `imq.user_repository.ldap.server` property to the port used by the LDAP server for SSL communication: for example,

  ```
  imq.user_repository.ldap.server=myhost:7878
  ```
  You will also need to activate SSL communication in the LDAP server.

In addition, you may need to edit the user and group names in the broker's access control file to match those defined in the LDAP user repository; see User Authorization for more information.

For example, to create administrative users, you use the access control file to specify those users and groups in the LDAP directory that can create ADMIN connections.

Any user or group that can create an ADMIN connection can issue administrative commands.

To Set Up an Administrative User

The following procedure makes use of a broker's access control file, which is described in User Authorization.

1. Enable the use of the access control file by setting the broker property `imq.accesscontrol.enabled` to `true`, which is the default value.

   The `imq.accesscontrol.enabled` property enables use of the access control file.

2. Open the access control file, `IMQ_VARHOME/instances/`*instanceName*`/etc/accesscontrol.properties`.

   The file contains an entry such as the following:

   ```
   service connection access control
   #################################
   connection.NORMAL.allow.user=*
   connection.ADMIN.allow.group=admin
   ```

   The entries listed are examples. Note that the `admin` group exists by default in the file-based user repository but does not exist by default in the LDAP directory.

3. To grant Message Queue administrator privileges to users, enter the user names as follows:

```
connection.ADMIN.allow.user= userName[[,userName2] …]
```

The users must be defined in the LDAP directory.

4. To grant Message Queue administrator privileges to groups, enter the group names as follows:

```
connection.ADMIN.allow.group= groupName[[,groupName2] …]
```

The groups must be defined in the LDAP directory.

Using JAAS-Based Authentication

The Java Authentication and Authorization Service (JAAS) API allows you to plug an external authentication mechanism into Message Queue. This section describes the information that the Message Queue message broker makes available to a JAAS-compliant authentication service and explains how to configure the broker to use such a service. The following sources provide further information on JAAS:

- For complete information about the JAAS API , see the *Java Authentication and Authorization Service (JAAS) Reference Guide*.

- For information about writing a JAAS login module, see the *Java Authentication and Authorization Service (JAAS) LoginModule Developer's Guide*.

JAAS is a core API in Java 2 Standard Edition (J2SE), and is therefore an integral part of Message Queue's runtime environment. It defines an abstraction layer between an application and an authentication mechanism, allowing the desired mechanism to be plugged in with no change to application code. In the case of the Message Queue service, the abstraction layer lies between the broker (application) and an authentication provider. By setting a few broker properties, it is possible to plug in any JAAS-compliant authentication service and to upgrade this service with no disruption or change to broker code.

> **Note:** You cannot use the Java Management Extensions (JMX) API to change JAAS-related broker properties. However, once JAAS-based authentication is configured, JMX client applications (like other clients) can be authenticated using this mechanism.

Elements of JAAS

Figure 9–2 shows the basic elements of JAAS: a JAAS client, a JAAS-compliant authentication service, and a JAAS configuration file.

- The JAAS client is an application wishing to perform authentication using a JAAS-compliant authentication service. The JAAS client communicates with the authentication service using one or more login modules and is responsible for providing a callback handler that the login module can call to obtain the user name, password, and other information needed for authentication.

- The JAAS-compliant authentication service consists of one or more login modules along with logic to perform the needed authentication. The login module (`LoginModule`) may include the authentication logic itself, or it may use a private protocol or API to communicate with an external security service that provides the logic.

■ The JAAS configuration file is a text file that the JAAS client uses to locate the login module(s) to be used.
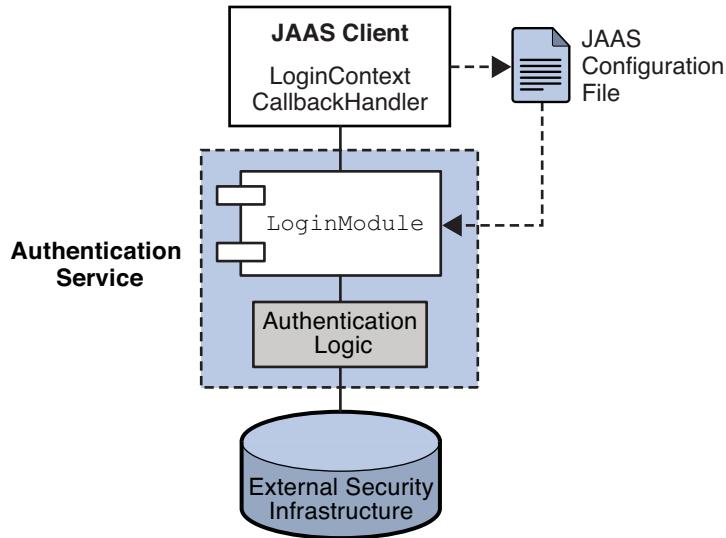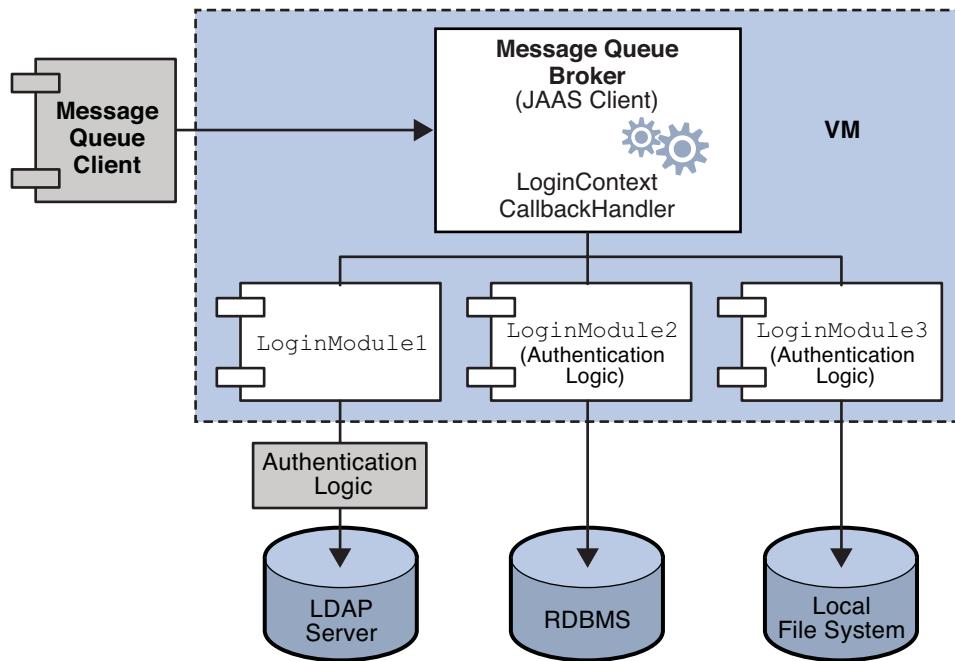
Figure 9–2   JAAS Elements

JAAS and Message Queue

Figure 9–3 shows how JAAS is used by the Message Queue broker. It shows a more complex implementation of the JAAS model shown in Figure 9–2.

Figure 9–3   How Message Queue Uses JAAS
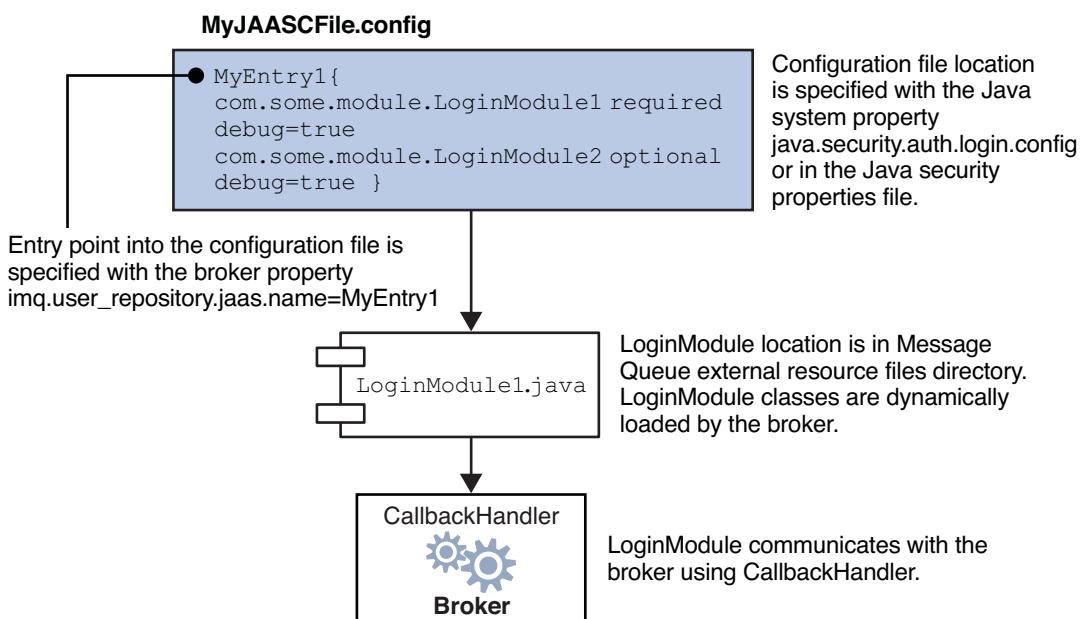
The authentication service layer, consisting of one or more login modules (if needed) and corresponding authentication logic, is separate from the broker. The login modules run in the same Java virtual machine as the broker. The broker is represented

to the login module as a login context, and communicates with the login module by means of a callback handler that is part of the broker runtime code.

The authentication service also supplies a JAAS configuration file containing entries that reference the login modules. The configuration file specifies the order in which the login modules (if more than one) are to be used and any conditions for their use. When the broker starts up, it locates the configuration file by consulting either the Java system property `java.security.auth.login.config` or the Java security properties file. The broker then selects an entry in the JAAS configuration file according to the value of the broker property `imq.user_repository.jaas.name`. That entry specifies which login module(s) will be used for authentication. The classes for the login modules are found in the Message Queue external resource files directory, `IMQ_HOMElib/ext`.

The relation between the configuration file, the login module, and the broker is shown in the following figure. Figure 9–4.

Figure 9–4   Setting Up JAAS Support

MyJAASCFile.config


```
MyEntry1{
  com.some.module.LoginModule1 required
  debug=true
  com.some.module.LoginModule2 optional
  debug=true }
```

Configuration file location is specified with the Java system property java.security.auth.login.config or in the Java security properties file.

Entry point into the configuration file is specified with the broker property imq.user_repository.jaas.name=MyEntry1

`LoginModule1.java`

LoginModule location is in Message Queue external resource files directory. LoginModule classes are dynamically loaded by the broker.

CallbackHandler

Broker

LoginModule communicates with the broker using CallbackHandler.

The fact that the broker uses a JAAS plug-in authentication service remains completely transparent to the Message Queue client. The client continues to connect to the broker as it did before, passing a user name and password. In turn, the broker uses a callback handler to pass login information to the authentication service, and the service uses the information to authenticate the user and return the results. If authentication succeeds, the broker grants the connection; if it fails, the client runtime returns a JMS security exception that the client must handle.

After the Message Queue client is authenticated, if there is further authorization to be done, the broker proceeds as it normally would, consulting the access control file to determine whether the authenticated client is authorized to perform the actions it undertakes: accessing a destination, consuming a message, browsing a queue, and so on.

Setting up JAAS-Compliant Authentication

Setting up JAAS-compliant authentication involves setting broker and system properties to select this type of authentication, to specify the location of the

configuration file, and to specify the entries to the login modules that are going to be used.

To set up JAAS support for Message Queue, you perform the following general steps. (These steps assume you are creating your own authentication service.)

1. Create one or more login module classes that implement the authentication service. The JAAS callback types that the broker supports are listed below.

   `javax.security.auth.callback.LanguageCallback`
   The broker uses this callback to pass the authentication service the locale in which the broker is running. This value can be used for localization.

   `javax.security.auth.callback.NameCallback`
   The broker uses this callback to pass to the authentication service the user name specified by the Message Queue client when the connection was requested.

   `javax.security.auth.callback.TextInputCallback`
   The broker uses this callback to pass the value of the following information to the login module (authentication service) when requested through the TextInputCallback.getPrompt() with the following strings:

   - `imq.authentication.type`: The broker authentication type in effect at runtime

   - `imq.accesscontrol.type`: The broker access control type in effect at runtime

   - `imq.authentication.clientip`: The client IP address (null if unavailable)

   - `imq.servicename`: The name of the connection service (`jms`, `ssljms`, `admin`, or `ssladmin`) being used by the client

   - `imq.servicetype`: The type of the connection service (`NORMAL` or `ADMIN`) being used by the client

   `javax.security.auth.callback.PasswordCallback`
   The broker uses this callback to pass to the authentication service the password specified by the Message Queue client when the connection was requested.

   `javax.security.auth.callback.TextOutputCallback`
   The broker handles this callback to provide logging service to the authentication service by logging the text output to the broker's log file. The callback's `MessageType ERROR, INFORMATION, WARNING` are mapped to the broker logging levels `ERROR, INFO, WARNING` respectively.

2. Create a JAAS configuration file with entries that reference the login module classes created in Step 1 and specify the location of this file.

3. Note the name of the entry in the JAAS configuration file (that references the login module implementation classes).

4. Archive the classes that implement the login modules to a jar file, and place the jar file in the Message Queue `lib/ext` directory.

5. Set the broker configuration properties that relate to JAAS support. These are described in Table 9–4.

6. Set the following system property (to specify the location of the JAAS configuration file).

   java.security.auth.login.config=*JAAS_Config_File_Location*

   For example, you can specify the location when you start the broker.

   imqbrokerd -Djava.security.auth.login.config=*JAAS_Config_File_Location*

There are other ways to specify the location of the JAAS configuration file. For additional information, see

http://download.oracle.com/javase/1.5.0/docs/guide/security/jaas/tutorials/LoginConfigFile.html

The following table lists the broker properties that need to be set to set up JAAS support.

Table 9–4   Broker Properties for JAAS Support

Property	Description
imq.authentication.type	Set to basic to indicate Base-64 password encoding. This is the only permissible value for JAAS authentication.
imq.authentication.basic.user_repository	Set to jaas to specify JAAS authentication.
imq.user_repository.jaas.name	Set to the name of the desired entry (in the JAAS configuration file) that references the login modules you want to use as the authentication mechanism. This is the name you noted inStep 3.
imq.user_repository.jaas.userPrincipalClass	This property, used by Message Queue access control, specifies the java.security.Principal implementation class in the login module(s) that the broker uses to extract the Principal name to represent the user entity in the Message Queue access control file. If, it is not specified, the user name passed from the Message Queue client when a connection was requested is used instead.
imq.user_repository.jaas.groupPrincipalClass	This property, used by Message Queue access control, specifies the java.security.Principal implementation class in the login module(s) that the broker uses to extract the Principal name to represent the group entity in the Message Queue access control file. If, it is not specified, the group rules, if any, in the Message Queue access control file are ignored.

User Authorization

An *access control file* contains rules that specify which users (or groups of users) are authorized to perform certain operations on a message broker. These operations include the following:

- Creating a connection

- Creating a message producer for a physical destination

- Creating a message consumer for a physical destination

- Browsing a queue destination

- Auto-creating a physical destination

If access control is enabled (that is, if the broker's imq.accesscontrol.enabled configuration property is set to true, the broker will consult its access control file whenever a client attempts one of these operations, to verify whether the user generating the request (or a group to which the user belongs) is authorized to perform the operation. By editing this file, you can restrict access to these operations to particular users and groups. Changes take effect immediately; there is no need to restart the broker after editing the file.

Access Control File Syntax

Each broker has it own access control file, created automatically when the broker is started. The file is named `accesscontrol.properties` and is located at `IMQ_VARHOME/instances/`*instanceName*`/etc`.

The file is formatted as a Java properties file. It starts with a `version` property defining the version of the file:

`version=JMQFileAccessControlModel/100`
This is followed by three sections specifying the access control for three categories of operations:

- Creating connections

- Creating message producers or consumers, or browsing a queue destination

- Auto-creating physical destinations

Each of these sections consists of a sequence of authorization rules specifying which users or groups are authorized to perform which specific operations. These rules have the following syntax:

resourceType.resourceVariant.operation.access.principalType=principals
Table 9–5 describes the various elements.

Table 9–5   Authorization Rule Elements

Element	Description
resourceType	Type of resource to which the rule applies: - `connection`: Connections - `queue`: Queue destinations - `topic`: Topic destinations
resourceVariant	Specific resource (connection service type or destination) to which the rule applies An asterisk (*) may be used as a wild-card character to denote all resources of a given type: for example, a rule beginning with `queue.*` applies to all queue destinations.
operation	Operation to which the rule applies This syntax element is not used for *resourceType*=`connection`.
access	Level of access authorized: - `allow`: Authorize user to perform operation - `deny`: Prohibit user from performing operation
principalType	Type of principal (user or group) to which the rule applies: - `user`: Individual user - `group`: User group
principals	List of principals (users or groups) to whom the rule applies, separated by commas An asterisk (*) may be used as a wild-card character to denote all users or all groups: for example, a rule ending with `user=*` applies to all users.

Example 9–3   Example 1

Rule: `queue.q1.consume.allow.user=*`

Description: allows all users to consume messages from the queue destination `q1`.

Example 9–4   Example 2

Rule: `queue.*.consume.allow.user=Snoopy`

Description: allows user `Snoopy` to consume messages from all queue destinations.

Example 9–5   Example 3

Rule: `topic.t1.produce.deny.user=Snoopy`

Description: prevents `Snoopy` from producing messages to the topic destination `t1`

> **Note:**   You can use Unicode escape (`\\u`*XXXX*) notation to specify non-ASCII user, group, or destination names. If you have edited and saved the access control file with these names in a non-ASCII encoding, you can use the Java `native2ascii` tool to convert the file to ASCII. See the *Java Internationalization FAQ* for more information.

Application of Authorization Rules

Authorization rules in the access control file are applied according to the following principles:

- Any operation not explicitly authorized through an authorization rule is implicitly prohibited. For example, if the access control file contains no authorization rules, all users are denied access to all operations.

- Authorization rules for specific users override those applying generically to all users. For example, the rules

  ```
  queue.q1.produce.allow.user=*
  queue.q1.produce.deny.user=Snoopy
  ```
  authorize all users except `Snoopy` to send messages to queue destination `q1`.

- Authorization rules for a specific user override those for any group to which the user belongs. For example, if user `Snoopy` is a member of group `user`, the rules

  ```
  queue.q1.consume.allow.group=user
  queue.q1.consume.deny.user=Snoopy
  ```
  authorize all members of `user` except `Snoopy` to receive messages from queue destination `q1`.

- Authorization rules applying generically to all users override those applying to all groups. For example, the rules

  ```
  topic.t1.produce.deny.group=*
  topic.t1.produce.allow.user=*
  ```
  authorize all users to publish messages to topic destination `t1`, overriding the rule denying such access to all groups.

- Authorization rules for specific resources override those applying generically to all resources of a given type. For example, the rules

  ```
  topic.*.consume.allow.user=Snoopy
  topic.t1.consume.deny.user=Snoopy
  ```
  authorize `Snoopy` to subscribe to all topic destinations except `t1`.

- Authorization rules authorizing and denying access to the same resource and operation for the same user or group cancel each other out, resulting in authorization being denied. For example, the rules

  ```
  queue.q1.browse.deny.user=Snoopy
  queue.q1.browse.allow.user=Snoopy
  ```

prevent `Snoopy` from browsing queue `q1`. The rules

```
topic.t1.consume.deny.group=user
topic.t1.consume.allow.group=user
```
prevent all members of group `user` from subscribing to topic `t1`.

- When multiple authorization rules are specified for the same resource, operation, and principal type, only the last rule applies. The rules

```
queue.q1.browse.allow.user=Snoopy,Linus
queue.q1.browse.allow.user=Snoopy
```
authorize user `Snoopy`, but not `Linus`, to browse queue destination `q1`.

Authorization Rules for Connection Services

Authorization rules with the resource type `connection` control access to the broker's connection services. The rule's *resourceVariant* element specifies the service type of the connection services to which the rule applies, as shown in Table 6–1; the only possible values are `NORMAL` or `ADMIN`. There is no *operation* element.

The default access control file contains the rules

```
connection.NORMAL.allow.user=*
connection.ADMIN.allow.group=admin
```
giving all users access to `NORMAL` connection services (`jms`, `ssljms`, `httpjms`, and `httpsjms`) and those in the `admin` group access to `ADMIN` connection services (`admin` and `ssladmin`). You can then add additional authorization rules to restrict the connection access privileges of specific users: for example, the rule

```
connection.NORMAL.deny.user=Snoopy
```
denies user `Snoopy` access privileges for connection services of type `NORMAL`.

If you are using a file-based user repository, the `admin` user group is created by the User Manager utility. If access control is disabled (`imq.accesscontrol.enabled = false`), all users in the `admin` group automatically have connection privileges for `ADMIN` connection services. If access control is enabled, access to these services is controlled by the authorization rules in the access control file.

If you are using an LDAP user repository, you must define your own user groups in the LDAP directory, using the tools provided by your LDAP vendor. You can either define a group named `admin`, which will then be governed by the default authorization rule shown above, or edit the access control file to refer to one or more other groups that you have defined in the LDAP directory. You must also explicitly enable access control by setting the broker's `imq.accesscontrol.enabled` property to `true`.

Authorization Rules for Physical Destinations

Access to specific physical destinations on the broker is controlled by authorization rules with a resource type of `queue` or `topic`, as the case may be. These rules regulate access to the following operations:

- Sending messages to a queue: produce operation

- Receiving messages from a queue: consume operation

- Publishing messages to a topic: produce operation

- Subscribing to and consuming messages from a topic: consume operation

- Browsing a queue: browse operation

By default, all users and groups are authorized to perform all of these operations on any physical destination. You can change this by editing the default authorization

rules in the access control properties file or overriding them with more specific rules of your own. For example, the rule

```
topic.Admissions.consume.deny.group=user
```
denies all members of the `user` group the ability to subscribe to the topic `Admissions`.

Authorization Rules for Auto-Created Physical Destinations

When a client creates a message producer or consumer for a physical destination that does not already exist, the broker will auto-create the destination (provided that the broker's `imq.autocreate.queue` or `imq.autocreate.topic` property is set to `true`).

The final section of the access control file controls the ability of users and groups to auto-create destinations, and to access any auto-created destinations. This is governed by authorization rules with a *resourceType* of `queue` or `topic` and an *operation* element of `create`. the *resourceVariant* element is omitted, since these rules apply to all auto-created queues or all auto-created topics, rather than any specific destination.

The default access control file contains the rules

```
queue.create.allow.user=*
topic.create.allow.user=*
```
authorizing all users to have physical destinations auto-created for them by the broker, and to have access to any auto-created destinations. You can edit the file to restrict such authorization for specific users. For example, the rule

```
topic.create.deny.user=Snoopy
```
denies user `Snoopy` the ability to auto-create topic destinations or to access any auto-created topic destinations.

> **Note:** The effect of such auto-creation rules must be congruent with that of other physical destination access rules. For example, if you change the destination authorization rule to prohibit any user from sending a message to a queue, but enable the auto-creation of queue destinations, the broker *will* create the physical destination if it does not exist, but will *not* deliver a message to it.

Message Encryption

This section explains how to set up a connection service based on the Secure Socket Layer (SSL) standard, which enables delivery of encrypted messages over the connection. Message Queue supports the following SSL-based connection services:

- The `ssljms` service delivers secure, encrypted messages between a client and a broker, using the TCP/IP transport protocol.

- The `httpsjms` service delivers secure, encrypted messages between a client and a broker, using an HTTPS tunnel servlet with the HTTP transport protocol.

- The `ssladmin` service creates a secure, encrypted connection between the Message Queue Command utility (`imqcmd`) and a broker, using the TCP/ IP transport protocol. Encrypted connections are not supported for the Administration Console (`imqadmin`).

- The `cluster` connection service is used internally to provide secure, encrypted communication between brokers in a cluster, using the TCP/IP transport protocol.

- A JMX connector that supports secure, encrypted communication between a JMX client and a broker's MBean server using the RMI transport protocol over TCP.

The remainder of this section describes how to set up secure connections over TCP/IP, using the `ssljms`, `ssladmin`, and `cluster` connection services. For information on setting up secure connections over HTTP with the `httpsjms` service, see HTTP/HTTPS Support.

Using Self-Signed Certificates

To use an SSL-based connection service over TCP /IP, you generate a public/private 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. This section describes how to set up an SSL-based service using such self-signed certificates.

For a stronger level of authentication, you can use signed certificates verified by a certification authority. The use of signed certificates involves some additional steps beyond those needed for self-signed certificates: you must first perform the procedures described in this section and then perform the additional steps in Using Signed Certificates.

Message Queue's support for SSL with self-signed certificates is oriented toward securing on-the-wire data, on the assumption that the client is communicating with a known and trusted server. Configuring SSL with self-signed certificates requires configuration on both the broker and client:

■   Setting Up an SSL-Based Connection Service Using Self-Signed Certificates

■   Configuring and Running an SSL-Based Client Using Self-Signed Certificates

Setting Up an SSL-Based Connection Service Using Self-Signed Certificates

The following sequence of procedures are needed to set up an SSL-based connection service for using self-signed certificates:

Note: Starting with release 4.0, the default value for the client connection factory property `imqSSLIsHostTrusted` is `false`. If your application depends on the prior default value of `true`, you need to reconfigure and to set the property explicitly to `true`. In particular, old or new clients using self-signed certificates should set this property to true; for example:

`java -DimqConnectionType=TLS -DimqSSLIsHostTrusted=true MyApp`

The administration tool `imqcmd` is also affected by this change. In addition to using the `-secure` option to specify that it uses a SSL-based `admin` connection service, the `imqSSLIsHostTrusted` should be set to `true` when connecting to a broker configured with a self-signed certificate. You can do this as follows:

`imqcmd list svc -secure -DimqSSLIsHostTrusted=true`

Alternatively, you can import the broker's self-signed certificate into the client runtime trust store. Use the procedure in To Install a Signed Certificate.

---

1.   Generate a self-signed certificate.

2. Enable the desired SSL-based connection services in the broker. These can include the `ssljms`, `ssladmin`, or `cluster` connection services.

3. Start the broker.

To Generate a Self-Signed Certificate Run the Key Tool utility (`imqkeytool`) to generate a self-signed certificate for the broker. (On Solaris and Linux operating systems, you may need to run the utility as the root user in order to have permission to create the keystore file.) The same certificate can be used for all SSL-based connection services (`ssljms`, `ssladmin`, `cluster` connection services, and the `ssljmxrmi` connector).

1. Enter the following at the command prompt:

   `imqkeytool broker`

   The Key Tool utility prompts you for a key store password:

2. At the prompt type a keystore password.

   The Keystore utility prompts you for identifying information from which to construct an X.500 distinguished name. The following table shows the prompts and the values to be provided for each. Values are case-insensitive and can include spaces.

Prompt	X.500 Attribute	Description	Example
What is your first and last name?	*commonName* (*CN*)	Fully qualified name of server running the broker	mqserver.sun.com
What is the name of your organizational unit?	*organizationalUnit* (*OU*)	Name of department or division	purchasing
What is the name of your organization?	*organizationName* (*ON*)	Name of larger organization, such as a company or government entity	Acme Widgets, Inc.
What is the name of your city or locality?	*localityName* (*L*)	Name of city or locality	San Francisco
What is the name of your state or province?	*stateName* (*ST*)	Full (unabbreviated) name of state or province	California
What is the two-letter country code for this unit?	*country* (*C*)	Standard two-letter country code	US

   The Key Tool utility displays the information you entered for confirmation. For example,

   ```
   Is CN=mqserver.sun.com, OU=purchasing, ON=Acme Widgets, Inc.,
   L=San Francisco, ST=California, C=US correct?
   ```

3. Accept the current values and proceed by typing `yes`.

   To reenter values, accept the default or enter `no`. After you confirm, the utility pauses while it generates a key pair.

   The utility asks for a password to lock the key pair (key password).

4. Press return.

This will set the same password for both the key password and the keystore password.

> **Caution:** Be sure to remember the password you specify. You must provide this password when you start the broker, to allow the broker to open the keystore file. You can store the keystore password in a password file (see Password Files).

The Key Tool utility generates a self-signed certificate and places it in Message Queue's keystore file. The keystore file is located in IMQ_HOME/etc by default.

The following are the configurable properties for the Message Queue keystore for SSL-based connection services:

imq.keystore.file.dirpath
Path to directory containing keystore file

imq.keystore.file.name
Name of key store file

imq.keystore.password
Ke store password (to be used only in a password file)

In some circumstances, you may need to regenerate a key pair in order to solve certain problems: for example, if you forget the key store password or if the SSL-based service fails to initialize when you start a broker and you get the exception:

```
java.security.UnrecoverableKeyException: Cannot recover key
```
(This exception may result if you provided a key password different from the keystore password when you generated the self-signed certificate.)

To Regenerate a Key Pair

1. Remove the broker's keystore file.

   The file is located in IMQ_HOME/etc by default.

2. Run imqkeytool again.

   The command will generate a new key pair, as described above.

To Enable an SSL-Based Connection Service in the Broker To enable an SSL-based connection service in the broker, you need to add the corresponding service or services to the imq.service.activelist property.

1. Open the broker's instance configuration file:

   IMQ_VARHOME/instances/*instanceName*/props/config.properties
2. Add an entry (if one does not already exist) for the imq.service.activelist property and include the desired SSL-based service(s) in the list.

   By default, the property includes the jms and admin connection services. Add the SSL-based service or services you wish to activate (ssljms, ssladmin, or both):

   ```
   imq.service.activelist=jms,admin,ssljms,ssladmin
   ```

> **Note:** The SSL-based `cluster` connection service is enabled using the `imq.cluster.transport` property rather than the `imq.service.activelist` property (see Cluster Connection Service Properties). To enable SSL for RMI-based JMX connectors, see SSL-Based JMX Connections.

3. Save and close the instance configuration file.

To Start the Broker Start the broker, providing the key store password.

> **Note:** When you start a broker or client with SSL, you may notice a sharp increase in CPU usage for a few seconds. This is because the JSSE (Java Secure Socket Extension) method `java.security.SecureRandom`, which Message Queue uses to generate random numbers, takes a significant amount of time to create the initial random number seed. Once the seed is created, the CPU usage level will drop to normal.

1. Start the broker, providing the keystore password.

Put the keystore password in a password file, as described in Password Files and set the `imq.passfile.enabled` property to `true`. You can now do one of the following:

- Pass the location of the password file to the `imqbrokerd` command:

  `imqbrokerd -passfile /`*passfileDirectory*`/`*passfileName*

- Start the broker without the `-passfile` option, but specify the location of the password file using the following two broker configuration properties:

  `imq.passfile.dirpath=/`*passfileDirectory*

  `imq.passfile.name=/`*passfileName*

2. If you are not using a password file, enter the keystore password at the prompt.

`imqbrokerd`

You are prompted for the keystore passwrd.

Configuring and Running an SSL-Based Client Using Self-Signed Certificates

The procedure for configuring a client to use an SSL-based connection service differs depending on whether it is an application client (using the `ssljms` connection service) or a Message Queue administrative client such as `imqcmd` (using the `ssladmin` connection service.)

Application Clients For application clients, you must make sure the client has the following `.jar` files specified in its `CLASSPATH` variable:

```
imq.jar
jms.jar
```
Once the `CLASSPATH` files are properly specified, one way to start the client and connect to the broker's `ssljms` connection service is by entering a command like the following:

```
java -DimqConnectionType=TLS clientAppName
```
This tells the connection to use an SSL-based connection service.

Administrative Clients For administrative clients, you can establish a secure connection by including the `-secure` option when you invoke the `imqcmd` command: for example,

```
imqcmd list svc  -b hostName:portNumber  -u userName  -secure
```
where *userName* is a valid `ADMIN` entry in the Message Queue user repository. The command will prompt you for the password.

Listing the connection services is a way to verify that the `ssladmin` service is running and that you can successfully make a secure administrative connection, as shown in Example 9–6.

Example 9–6   Connection Services Listing

```
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.
```

Using Signed Certificates

Signed certificates provide a stronger level of server authentication than self-signed certificates. You can implement signed certificates only between a client and broker, and currently not between multiple brokers in a cluster. This requires the following extra procedures in addition to the ones described in Using Self-Signed Certificates. Using signed certificates requires configuration on both the broker and client:

- Obtaining and Installing a Signed Certificate
- Configuring the Client to Require Signed Certificates

Obtaining and Installing a Signed Certificate

The following procedures explain how to obtain and install a signed certificate.

To Obtain a Signed Certificate

1. Use the J2SE `keytool` command to generate a certificate signing request (CSR) for the self-signed certificate you generated in the preceding section.

   Information about the `keytool` command can be found at

   - http://download.oracle.com/javase/1.5.0/docs/tooldocs/solaris/keytool.html

   Here is an example:

   ```
   keytool  -certreq  -keyalg RSA  -alias imq  -file certreq.csr
   -keystore /etc/imq/keystore  -storepass  myStorePassword
   ```
   This generates a CSR encapsulating the certificate in the specified file (`certreq.csr` in the example).

2. Use the CSR to generate or request a signed certificate.

You can do this by either of the following methods:

- Have the certificate signed by a well known certification authority (CA), such as Thawte or Verisign. See your CA's documentation for more information on how to do this.

- Sign the certificate yourself, using an SSL signing software package.

  The resulting signed certificate is a sequence of ASCII characters. If you receive the signed certificate from a CA, it may arrive as an e-mail attachment or in the text of a message.

3. Save the signed certificate in a file.

   The instructions below use the example name `broker.cer` to represent the broker certificate.

To Install a Signed Certificate

1. Check whether J2SE supports your certification authority by default.

   The following command lists the root CAs in the system key store:

   ```
   keytool  -v  -list  -keystore $JAVA_HOME/lib/security/cacerts
   ```
   If your CA is listed, skip the next step.

2. If your certification authority is not supported in J2SE, import the CA's root certificate into the Message Queue key store.

   Here is an example:

   ```
   keytool  -import  -alias ca  -file ca.cer  -noprompt  -trustcacerts
   -keystore /etc/imq/keystore  -storepass  myStorePassword
   ```
   where `ca.cer` is the file containing the root certificate obtained from the CA.

   If you are using a CA test certificate, you probably need to import the test CA root certificate. Your CA should have instructions on how to obtain a copy.

3. Import the signed certificate into the key store to replace the original self-signed certificate.

   Here is an example:

   ```
   keytool  -import  -alias imq  -file broker.cer  -noprompt  -trustcacerts
   -keystore /etc/imq/keystore  -storepass  myStorePassword
   ```
   where `broker.cer` is the file containing the signed certificate that you received from the CA.

   The Message Queue key store now contains a signed certificate to use for SSL connections.

Configuring the Client to Require Signed Certificates

You must now configure the Message Queue client runtime to require signed certificates, and ensure that it trusts the certification authority that signed the certificate.

> **Note:** By default, starting with release 4.0, the connection factory object that the client will be using to establish broker connections has its `imqSSLIsHostTrusted` attribute set to `false`, meaning that the client runtime will attempt to validate all certificates. Validation will fail if the signer of the certificate is not in the client's trust store.

To Configure the Client Runtime to Require Signed Certificates

1. Verify whether the signing authority is registered in the client's trust store.

    To test whether the client will accept certificates signed by your certification authority, try to establish an SSL connection, as described above under Configuring and Running an SSL-Based Client Using Self-Signed Certificates. If the CA is in the client's trust store, the connection will succeed and you can skip the next step. If the connection fails with a certificate validation error, go on to the next step.

2. Install the signing CA's root certificate in the client's trust store.

    The client searches the key store files `cacerts` and `jssecacerts` by default, so no further configuration is necessary if you install the certificate in either of those files. The following example installs a test root certificate from the Verisign certification authority from a file named `testrootca.cer` into the default system certificate file, `cacerts`. The example assumes that J2SE is installed in the directory `$JAVA_HOME/usr/j2se`:

    ```
    keytool  -import  -keystore /usr/j2se/jre/lib/security/cacerts
    -alias VerisignTestCA  -file testrootca.cer  -noprompt
    -trustcacerts  -storepass myStorePassword
    ```
    An alternative (and recommended) option is to install the root certificate into the alternative system certificate file, `jssecacerts`:

    ```
    keytool  -import  -keystore /usr/j2se/jre/lib/security/jssecacerts
    -alias VerisignTestCA  -file testrootca.cer  -noprompt
    -trustcacerts  -storepass myStorePassword
    ```
    A third possibility is to install the root certificate into some other key store file and configure the client to use that as its trust store. The following example installs into the file `/home/smith/.keystore`:

    ```
    keytool  -import  -keystore /home/smith/.keystore
    -alias VerisignTestCA  -file testrootca.cer  -noprompt
    -trustcacerts  -storepass myStorePassword
    ```
    Since the client does not search this key store by default, you must explicitly provide its location to the client to use as a trust store. You do this by setting the Java system property `javax.net.ssl.trustStore` once the client is running:

    ```
    javax.net.ssl.trustStore=/home/smith/.keystore
    ```

Password Files

Several types of command require passwords. In Table 9–6, the first column lists the commands that require passwords and the second column lists the reason that passwords are needed.

Table 9–6   Commands That Use Passwords

Command	Description	Purpose of Password
`imqbrokerd`	Start broker	Access a JDBC-based persistent data store, an SSL certificate key store, or an LDAP user repository
`imqcmd`	Manage broker	Authenticate an administrative user who is authorized to use the command
`imqdbmgr`	Manage JDBC-based data store	Access the data store

You can specify these passwords in a password file and use the `-passfile` option to specify the name of the file. This is the format for the `-passfile` option:

```
imqbrokerd  -passfile filePath
```

> **Note:** In previous versions of Message Queue, you could use the `-p`, `-password`, `-dbpassword`, and `-ldappassword` options to specify passwords on the command line. As of Message Queue 4.0, these options are deprecated and are no longer supported; you must use a password file instead.

Security Considerations for Passwords and Password Files

Typing a password interactively, in response to a prompt, is the most secure method of specifying a password (provided that your monitor is not visible to other people). You can also specify a password file on the command line. For non-interactive use of commands, however, you must use a password file.

A password file that contains clear passwords should be obfusticated using `imqusermgr encode`. An obfusticated password file must be stored in location that allows you to set directory and file permissions to protect them from unauthorized access. Set the permissions so that they limit the users who can view the file, but provide read access to the user who starts the broker.

> **Note:** Oracle recommends using `imqusermgr encode` to obfuscate passfiles. See Secure Password Files.

Password File Contents

A password file is a simple text file containing a set of properties and values. Each value is a password used by a command. Table 9–7 shows the types of passwords that a password file can contain.

Table 9–7   Passwords in a Password File

Password	Affected Commands	Description
`imq.imqcmd.password`	`imqcmd`	Administrator password for Message Queue Command utility (authenticated for each command)
`imq.keystore.password`	`imqbrokerd`	Key store password for SSL-based services
`imq.persist.jdbc.password`	`imqbrokerd` `imqdbmgr`	Password for opening a database connection, if required
`imq.user_repository.ldap.password`	`imqbrokerd`	Password associated with the distinguished name assigned to a broker for binding to a configured LDAP user repository

A sample password file, `IMQ_HOME/etc/passfile.sample`, is provided as part of your Message Queue installation.

Secure Password Files

As of Message Queue 5.0, Oracle provides User Manager utility (`imqusermgr`) commands to obfuscate (convert to an unreadable format) password files (passfiles).

Obfuscate a Passfile

To obfuscate a password file:

1. Place all password property *name=value* pairs in a password file. See Password File Contents.

2. Use `imqusermgr encode` to obfuscate the password file to a new password file. See User Manager Utility.

   For example: `imqusermgr encode -src IMQ_HOME/etc/passfile.clear -target IMQ_HOME/etc/passfile.secure`

3. Use the new password file for commands that require those passwords. For example:

   `imqbrokerd -passfile IMQ_HOME/etc/passfile.secure`

Add a Password to a Password File

To add a new password to an existing password file:

1. Add the new *name=value* pair to an existing passfile (clear text or obfuscated).

2. Use `imqusermgr encode` to obfuscate the password file. See Obfuscate a Passfile.

Delete a Password from a Password File

To delete an existing password from a passfile (clear text or obfuscated), delete the line containing the password and save the passfile.

Change a Password in a Password File

To change a password in an obfuscated passfile:

1. Delete the existing password from the passfile (clear text or obfuscated).

2. Add a new line containing the new password.

3. Use `imqusermgr encode` to obfuscate the password file. See Obfuscate a Passfile.

Deobfuscate a Password File

To convert an obfuscated passfile to clear text, use `imqusermgr decode`. See User Manager Utility. For example:

`imqusermgr decode -src IMQ_HOME/etc/passfile.secure -target IMQ_HOME/etc/passfile.clear`

Upgrading Clear Text Password Files

Although using clear text passfiles is still supported in this release of Message Queue, Oracle recommends existing clear text passfiles be obfuscated by running `imqusermge encode`. See User Manager Utility.

A broker logs a warning message when a clear text password is found in a password file.

Connecting Through a Firewall

When a client application is separated from the broker by a firewall, special measures are needed in order to establish a connection. One approach is to use the `httpjms` or `httpsjms` connection service, which can "tunnel" through the firewall; see HTTP/HTTPS Support for details. HTTP connections are slower than other connection

services, however; a faster alternative is to bypass the Message Queue Port Mapper and explicitly assign a static port address to the desired connection service, and then open that specific port in the firewall. This approach can be used to connect through a firewall using the `jms` or `ssljms` connection service (or, in unusual cases, `admin` or `ssladmin`).

Table 9–8   Broker Configuration Properties for Static Port Addresses

Connection Service	Configuration Property
jms	imq.jms.tcp.port
ssljms	imq.ssljms.tls.port
admin	imq.admin.tcp.port
ssladmin	imq.ssladmin.tls.port

To Enable Broker Connections Through a Firewall

1. Assign a static port address to the connection service you wish to use.

    To bypass the Port Mapper and assign a static port number directly to a connection service, set the broker configuration property imq.*serviceName*.*protocolType*.port, where *serviceName* is the name of the connection service and*protocolType* is its protocol type (see Table 9–8). As with all broker configuration properties, you can specify this property either in the broker's instance configuration file or from the command line when starting the broker. For example, to assign port number 10234 to the jms connection service, either include the line

    ```
    imq.jms.tcp.port=10234
    ```
    in the configuration file or start the broker with the command

    ```
    imqbrokerd  -name brokerName  -Dimq.jms.tcp.port=10234
    ```
    where *brokerName* is the name of the broker to be started.

2. Configure the firewall to allow connections to the port number you assigned to the connection service.

    You must also allow connections through the firewall to Message Queue's Port Mapper port (normally 7676, unless you have reassigned it to some other port). In the example above, for instance, you would need to open the firewall for ports 10234 and 7676.

Audit Logging with the Solaris BSM Audit Log

Message Queue supports audit logging. When audit logging is enabled, Message Queue generates a record for the following types of events:

- Startup, shutdown, restart, and removal of a broker instance

- User authentication and authorization

- Reset of a persistent store

- Creation, purge, and destruction of a physical destination

- Administrative destruction of a durable subscriber

Message Queue supports logging audit records to the Message Queuebroker log file and to the Solaris BSM audit log:

- To log audit records to the Message Queue broker log file, set the `imq.audit.enabled` broker property to `true`. All audit records in the broker log contain the keyword `AUDIT`.

- To log audit records to the Solaris BSM audit log, set the `imq.audit.bsm.disabled` broker property to `false`.

> **Note:** To log audit records to the Solaris BSM audit log, you must run the broker as `root`, and `/usr/lib/audit/Audit.jar` must be in the broker classpath.

10

Configuring and Managing Broker Clusters

Message Queue supports the use of broker clusters: groups of brokers working together to loprovide message delivery services to clients. Clusters enable a message service to scale its operations to meet an increasing volume of message traffic by distributing client connections among multiple brokers.

In addition, clusters provide for message service availability. In the case of a conventional cluster, if a broker fails, clients connected to that broker can reconnect to another broker in the cluster and continue producing and consuming messages. In the case of an enhanced cluster, if a broker fails, clients connected to that broker reconnect to a failover broker that takes over the pending work of the failed broker, delivering messages without interruption of service.

See "Broker Clusters" in *Open Message Queue Technical Overview* for a description of conventional and enhanced broker clusters and how they operate.

This chapter describes how to configure and manage both conventional and enhanced broker clusters:

- Configuring Broker Clusters
- Managing Broker Clusters

Configuring Broker Clusters

You create a broker cluster by specifying *cluster configuration properties* for each of its member brokers. Except where noted in this chapter, cluster configuration properties must be set to the same value for each broker in a cluster. This section introduces these properties and the use of a cluster configuration file to specify them:

- The Cluster Configuration File
- Cluster Configuration Properties
- Displaying a Cluster Configuration

The Cluster Configuration File

Like all broker properties, cluster configuration properties can be set individually for each broker in a cluster, either in its instance configuration file (`config.properties`) or by using the `-D` option on the command line when you start the broker. However, except where noted in this chapter, each cluster configuration property must be set to the same value for each broker in a cluster.

For example, to specify the transport protocol for the cluster connection service, you can include the following property in the instance configuration file for each broker in the cluster: `imq.cluster.transport=ssl`. If you need to change the value of this

property, you must change its value in the instance configuration file for every broker in the cluster.

For consistency and ease of maintenance, it is generally more convenient to collect all of the common cluster configuration properties into a central cluster configuration file that all of the individual brokers in a cluster reference. Using a cluster configuration file prevents the settings from getting out of synch and ensures that all brokers in a cluster use the same, consistent cluster configuration information.

When using a cluster configuration file, each broker's instance configuration file must point to the location of the cluster configuration file by setting the `imq.cluster.url` property. For example,

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

> **Note:** A cluster configuration file can also include broker properties that are not used specifically for cluster configuration. For example, you can place any broker property in the cluster configuration file that has the same value for all brokers in a cluster. For more information, see Connecting Brokers into a Conventional Cluster

Cluster Configuration Properties

This section reviews the most important cluster configuration properties, grouped into the following categories:

- Cluster Connection Service Properties
- Conventional Broker Cluster Properties
- Enhanced Broker Cluster Properties

A complete list of cluster configuration properties can be found in Table 17–14

Cluster Connection Service Properties

The following properties are used to configure the cluster connection service used for internal communication between brokers in the cluster. These properties are used by both conventional and enhanced clusters.

- **imq.cluster.transport** specifies the transport protocol used by the `cluster` connection service, such as `tcp` or `ssl`.

- **imq.cluster.port** specifies the port number for the `cluster` connection service. You might need to set this property, for instance, to specify a static port number for connecting to the broker through a firewall.

- **imq.cluster.hostname** specifies the host name or IP address for the `cluster` connection service, used for internal communication between brokers in the cluster. The default setting works fine, however, explicitly setting the property can be useful if there is more than one network interface card installed in a computer. If you set the value of this property to `localhost`, the value will be ignored and the default will be used.

Conventional Broker Cluster Properties

In addition to the properties listed in Cluster Connection Service Properties, all conventional clusters use the following properties:

- **imq.cluster.brokerlist** specifies a list of broker addresses defining the membership of the cluster; all brokers in the cluster must have the same value for this property.

  For example, to create a conventional cluster consisting of brokers at port `9876` on `host1`, port `5000` on `host2`, and the default port (`7676`) on `ctrlhost`, use the following value:

  `imq.cluster.brokerlist=host1:9876,host2:5000,ctrlhost`

- **imq.cluster.nomasterbroker** specifies whether the cluster is a conventional cluster of peer brokers, which uses a shared JDBC data store for the cluster's configuration change record. When `true`, the cluster is a conventional cluster of peer brokers. When `false` (or omitted, as `false` is the default), the cluster is considered to be a conventional cluster with master broker, even if no master broker is actually specified. All brokers in a given cluster must have the same value for this property.

Each type of conventional cluster has additional properties to support its configuration, as described in the following two sections.

Additional Properties for Conventional Clusters with Master Broker The following additional properties are used to configure a conventional cluster with a master broker:

- **imq.cluster.masterbroker** specifies which broker in a conventional cluster is the master broker that maintains the configuration change record that tracks the addition and deletion of destinations and durable subscribers. For example:

  `imq.cluster.masterbroker=host2:5000`

  While specifying a master broker using the `imq.cluster.masterbroker` is not mandatory for a conventional cluster with master broker to function, it guarantees that persistent information propagated across brokers (destinations and durable subscriptions) is always synchronized. See "Conventional Clusters" in *Open Message Queue Technical Overview*.

- **imq.cluster.dynamicChangeMasterBrokerEnabled** specifies whether the master broker can be changed to another broker in the cluster without stopping all the broker in the cluster. All brokers in a given cluster must have the same value for this property.

Additional Properties for Conventional Clusters of Peer Brokers The following additional properties are used to configure a conventional cluster of peer brokers. All brokers in a given cluster must have the same values for these properties.

- **imq.cluster.clusterid** specifies the cluster identifier, which will be appended to the name of the configuration change record's database table in the JDBC data store. The value of this property must be the same for all brokers in a given cluster, but must be unique for each cluster: no two clusters may have the same cluster identifier.

- **imq.cluster.sharecc.persist.jdbc.dbVendor** specifies the name of the database vendor of the JDBC data store housing the configuration change record's table.

- **imq.cluster.sharecc.persist.jdbc.<vendorName>.user** specifies the user name, if required, for connecting to the database from vendor *<vendorName>*.

- **imq.cluster.sharecc.persist.jdbc.<vendorName>.needpassword** specifies whether a password is needed for connecting to the database from vendor *<vendorName>*.

- **imq.cluster.sharecc.persist.jdbc.<vendorName>.password** specifies the password, if required, for connecting to the database from vendor *<vendorName>*. This value should be set only in password files, as described in Password Files.

- **imq.cluster.sharecc.persist.jdbc.<vendorName>.driver** specifies the Java class name of the JDBC driver, if required, for connecting to the database from vendor *<vendorName>*.

- **imq.cluster.sharecc.persist.jdbc.<vendorName>.opendburl** specifies the URL for connecting to an existing database from vendor *<vendorName>*. This applies when a `java.sql.Driver` is used to connect to the database.

- **imq.cluster.sharecc.persist.jdbc.<vendorName>.createdburl** optionally specifies the URL for creating a new database from vendor *<vendorName>*. This applies only to embedded databases, such as Java DB.

- **imq.cluster.sharecc.persist.jdbc.<vendorName>.closedburl** optionally specifies the URL for closing a connection to the database from vendor *<vendorName>*. This applies only to some embedded databases, such as Java DB.

- **imq.cluster.sharecc.persist.jdbc.<vendorName>.tableoption** optionally specifies vendor-specific options to be passed to the database from vendor *<vendorName>* when creating the table schema.

- **imq.cluster.sharecc.persist.jdbc.<vendorName>.property.<propName>** specifies a vendor-specific property *<propName>* for the database from vendor *<vendorName>*.

Enhanced Broker Cluster Properties

Enhanced broker clusters, which share a JDBC-based data store, require more configuration than do conventional broker clusters. In addition to the properties listed in Cluster Connection Service Properties, the following categories of properties are used to configure an enhanced cluster:

- Enhanced Clusters: General Configuration Properties

- Enhanced Clusters: JDBC Configuration Properties

- Enhanced Clusters: Failure Detection Properties

Enhanced Clusters: General Configuration Properties

- **imq.cluster.ha** is a boolean value that specifies if the cluster is an enhanced cluster (`true`) or a conventional broker (`false`). The default value is `false`.

  If set to `true`, mechanisms for failure detection and takeover of a failed broker are enabled. Enhanced clusters are self-configuring: any broker configured to use the cluster's shared data store is automatically registered as part of the cluster, without further action on your part. If the conventional cluster property, `imq.cluster.brokerlist`, is specified for a high-availability broker, the property is ignored and a warning message is logged at broker startup.

- **imq.persist.store** specifies the model for a broker's persistent data store. This property must be set to the value `jdbc` for every broker in an enhanced cluster.

- **imq.cluster.clusterid** specifies the cluster identifier, which will be appended to the names of all database tables in the cluster's shared persistent store. The value of this property must be the same for all brokers in a given cluster, but must be unique for each cluster: no two running clusters may have the same cluster identifier.

- **imq.brokerid** is a broker identifier that must be unique for each broker in the cluster. Hence, this property must be set in each broker's instance configuration file rather than in a cluster configuration file.

Enhanced Clusters: JDBC Configuration Properties The persistent data store for an enhanced cluster is maintained on a highly-available JDBC database.

The highly-availabile database may be MySQL Cluster Edition or Oracle Real Application Clusters (RAC), or it may be an open-source or third-party product. As described in JDBC-Based Persistence Properties, the `imq.persist.jdbc.dbVendor` broker property specifies the name of the database vendor, and all of the remaining JDBC-related properties are qualified with this vendor name.

The JDBC-related properties are discussed under JDBC-Based Persistence Properties and summarized in Table 17–8. See the example configuration for MySQL in Example 8–1.

> **Note:**   In setting JDBC-related properties for an enhanced cluster when using MySQL Cluster Edition as a highly-available database, you must specify the NDB Storage Engine rather than the InnoDB Storage Engine set by Message Queue by default. To specify the NDB Storage Engine, set the following broker property for all brokers in the cluster:
>
> `imq.persist.jdbc.mysql.tableoption=ENGINE=NDBCLUSTER`

Enhanced Clusters: Failure Detection Properties The following configuration properties (listed in Table 17–14) specify the parameters for the exchange of heartbeat and status information within an enhanced cluster:

- **imq.cluster.heartbeat.hostname** specifies the host name (or IP address) for the heartbeat connection service.

- **imq.cluster.heartbeat.port** specifies the port number for the heartbeat connection service.

- **imq.cluster.heartbeat.interval** specifies the interval, in seconds, at which heartbeat packets are transmitted.

- **imq.cluster.heartbeat.threshold** specifies the number of missed heartbeat intervals after which a broker is considered suspect of failure.

- **imq.cluster.monitor.interval** specifies the interval, in seconds, at which to monitor a suspect broker's state information to determine whether it has failed.

- **imq.cluster.monitor.threshold** specifies the number of elapsed monitor intervals after which a suspect broker is considered to have failed.

Smaller values for these heartbeat and monitoring intervals will result in quicker reaction to broker failure, but at the cost of reduced performance and increased likelihood of false suspicions and erroneous failure detection.

Displaying a Cluster Configuration

To display information about a cluster's configuration, use the Command utility's `list bkr` subcommand:

```
imqcmd list bkr
```
This lists the current state of all brokers included in the cluster to which a given broker belongs. The broker states are described in the following table:

Table 10–1   Broker States

Broker State	Meaning
OPERATING	Broker is operating
TAKEOVER_STARTED	For enhanced clusters, broker has begun taking over persistent data store from another broker
TAKEOVER_COMPLETE	For enhanced clusters, broker has finished taking over persistent data store from another broker
TAKEOVER_FAILED	For enhanced clusters, attempted takeover has failed
QUIESCE_STARTED	Broker has begun quiescing
QUIESCE_COMPLETE	Broker has finished quiescing
SHUTDOWN_STARTED	Broker has begun shutting down
BROKER_DOWN	Broker is down
UNKNOWN	Broker state unknown

The results of the `imqcmd list bkr` command are shown in (for a conventional cluster) and (for an enhanced cluster).

Example 10–1   Configuration Listing for a Conventional Cluster

```
Listing all the brokers in the cluster that the following broker is a member of:

------------------------
Host        Primary Port
------------------------
localHost   7676

Cluster Is Highly Available            False

------------------------
Address         State
--------------------
whippet:7676    OPERATING
greyhound:7676  OPERATING
```

Example 10–2   Configuration Listing for an Enhanced Cluster

```
Listing all the brokers in the cluster that the following broker is a member of:

----------------------------------------------
Host        Primary Port    Cluster Broker ID
----------------------------------------------
localHost   7676            brokerA

Cluster ID                         myClusterID
Cluster Is Highly Available        True

--------------------------------------------------------------------------------
----------------------------
                                                                      ID of
broker      Time since last
Broker ID      Address         State                 Msgs in store   performing
```

```
takeover    status timestamp
-----------------------------------------------------------------------------------
----------------------------
brokerA        localhost:7676  OPERATING            121
30 sec
brokerB        greyhound:7676  TAKEOVER_STARTED      52            brokerA
3 hrs
brokerC        jpgserv:7676    SHUTDOWN_STARTED      12346
10 sec
brokerD        icdev:7676      TAKEOVER_COMPLETE     0             brokerA
2 min
brokerE        mrperf:7676     *unknown             12
0 sec
brokerG        iclab1:7676     QUIESCING             4
2 sec
brokerH        iclab2:7676     QUIESCE_COMPLETE      8
5 sec
```

Managing Broker Clusters

The following sections describe how to perform various administrative management tasks for conventional and enhanced clusters, respectively.

- Managing Conventional Clusters

- Managing Enhanced Clusters

- Converting a Conventional Cluster to an Enhanced Cluster

Managing Conventional Clusters

The procedures in this section show how to perform the following tasks for a conventional cluster:

- Connecting Brokers into a Conventional Cluster

- Adding Brokers to a Conventional Cluster

- Removing Brokers From a Conventional Cluster

- Changing the Master Broker in a Conventional Cluster with Master Broker

- Managing a Conventional Cluster's Configuration Change Record

- Converting Between Types of Conventional Clusters

Connecting Brokers into a Conventional Cluster

There are two general methods of connecting brokers into a conventional cluster: from the command line (using the -cluster option) or by setting the imq.cluster.brokerlist property in the cluster configuration file.

Whichever method you use, each broker that you start attempts to connect to the other brokers in the cluster every five seconds until the connection succeeds.

For a cluster configured with master broker, the connection will succeed once the master broker is started up (if one is configured). If a broker in the cluster starts before the master broker, it will remain in a suspended state, rejecting client connections, until the master broker starts; the suspended broker then will automatically become fully functional. It is therefore a good idea to start the master broker first and then the others, after the master broker has completed its startup.

When connecting brokers into a conventional cluster, you should be aware of the following issues:

- **Mixed broker versions.** A conventional cluster can contain brokers of different versions if all brokers have a version at least as great as that of the master broker. If the cluster is not configured to use a master broker, then all brokers must be of the same version.

- **Matching broker property values.** In addition to cluster configuration properties, the following broker properties also must have the same value for all brokers in a cluster:

    - imq.service.activelist

    - imq.autocreate.queue

    - imq.autocreate.topic

    - imq.autocreate.queue.maxNumActiveConsumers

    - imq.autocreate.queue.maxNumBackupConsumers

    This restriction is particularly important when a cluster contains mixed broker versions that might contain properties with different default values. For example, If you are clustering a Message Queue version 4.1 or later broker together with those from earlier versions than Message Queue 4.1, you must set the value of the `imq.autocreate.queue.maxNumActiveConsumers` property, which has different default values before and after version 4.1 (`1` and `-1`, respectively), to be the same. Otherwise the brokers will not be able to establish a cluster connection.

- **Multiple interface cards**. On a multi-homed computer, in which there is more than one network interface card, be sure to explicitly set the network interface to be used by the broker for client connection services (`imq.hostname`) *and* for the cluster connection service (`imq.cluster.hostname`). Setting the `imq.hostname` value also effectively sets the value for other properties that use `imq.hostname` as their default value, such as `imq.portmapper.hostname`, `imq.cluster.hostname`, and so on. If `imq.cluster.hostname` is not set, then connections between brokers might not succeed and as a result, the cluster will not be established.

- **Network loopback IP address.** You must make sure that no broker in the cluster is given an address that resolves to a loopback network (`127.*.*.*`) IP address. Any broker configured with such an address will be unable to connect to other brokers in the cluster.

    In particular, some Linux installers automatically set the local host to a loopback network address, most commonly `127.0.0.1`. On such systems, you must do the following: For each Linux system participating in the cluster, check the `/etc/hosts` file as part of cluster setup. If the system uses a static IP address, edit the `/etc/hosts` file to specify the correct address for the local host. If the address is registered with Domain Name Service (DNS), edit the file `/etc/nsswitch.conf` so that DNS lookup is performed before consulting the local `hosts` file.

To Connect Brokers Using a Cluster Configuration File The method best suited for production systems is to use a cluster configuration file to specify the configuration of the cluster:

1. If using a conventional cluster of peer brokers, configure the use of the shared JDBC data store for the configuration change record:

    - Use the `imqdbmgr create sharecc_tbl` command to create the database table for the configuration change record.

   ■   Place a copy of, or a symbolic link to, your JDBC driver's `.jar` file in the
       Message Queue external resource files directory, `IMQ_HOME/lib/ext`, on each
       host where a broker will run.

2. Create a cluster configuration file that uses the `imq.cluster.brokerlist` property
   to specify the list of brokers to be connected.

   If you are using a master broker, identify it with the `imq.cluster.masterbroker`
   property in the configuration file.

   If you are using a cluster of peer brokers, set the `imq.cluster.nomasterbroker`
   property to `true`, and set `imq.cluster.sharecc.persist.jdbc.*` properties as
   appropriate in the configuration file.

3. For each broker in the cluster, set the `imq.cluster.url` property in the broker's
   instance configuration file to point to the cluster configuration file.

4. Use the `imqbrokerd` command to start each broker.

   If there is a master broker, start it first, then the others after it has completed its
   startup.

To Connect Brokers from the Command Line Connecting brokers to a cluster from the
command line involves starting each broker with the `imqbrokerd` command using the
`-cluster` option to specify the complete list of brokers to be included in the cluster.

For example, the following command starts a broker as part of a cluster consisting of
the brokers running at the default port (`7676`) on `host1`, at port `5000` on `host2`, and at
port `9876` on the default host (`localhost`):

```
imqbrokerd -cluster host1,host2:5000,:9876
```

The value specified for the `-cluster` option must be the same for all brokers in the
cluster.

Before You Begin

Set any necessary broker properties, except `imq.cluster.brokerlist`, in each broker's
configuration file before performing the following procedure.

1. If using a conventional cluster of peer brokers:

   a. Configure the use of the shared JDBC data store for the configuration change
      record:

      –   Use the `imqdbmgr create sharecc_tbl` command to create the database
          table for the configuration change record.

      –   Place a copy of, or a symbolic link to, your JDBC driver's `.jar` file in the
          Message Queue external resource files directory, `IMQ_HOME/lib/ext`, on
          each host where a broker will run.

   b. Start each broker in the cluster with the `imqbrokerd` command, specifying in
      the `-cluster` option the same complete list of brokers.

2. If using a conventional cluster with master broker:

   a. Start the master broker with the `imqbrokerd` command, specifying in the
      `-cluster` option the complete list of brokers.

   b. Once the master broker is running, start each of the other brokers in the cluster
      with the `imqbrokerd` command, specifying in the `-cluster` option the same
      complete list of brokers as you used to start the master broker.

To Establish Secure Connections Between Brokers If you want secure, encrypted message delivery between brokers in a cluster, configure the `cluster` connection service to use an SSL-based transport protocol:

1. For each broker in the cluster, set up SSL-based connection services, as described in Message Encryption.

2. Set each broker's `imq.cluster.transport` property to `ssl`, either in the cluster configuration file or individually for each broker.

Adding Brokers to a Conventional Cluster

The procedure for adding a new broker to a conventional cluster depends on whether the cluster uses a cluster configuration file.

To Add a New Broker to a Conventional Cluster 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 each broker to reload the `imq.cluster.brokerlist` property. It is not necessary to issue this command to every broker in the cluster; executing it for any one broker will cause all of them to reload the cluster configuration.

3. *(Optional)* Set the value of the `imq.cluster.url` property in the new broker's instance configuration file (`config.properties`) to point to the cluster configuration file.

4. Start the new broker.

    If you did not perform step 3, use the `-D` option on the `imqbrokerd` command line to set the value of `imq.cluster.url` to the location of the cluster configuration file.

To Add a New Broker to a Conventional Cluster Without a Cluster Configuration File

1. *(Optional)* Set the values of the following properties in the new broker's instance configuration file (`config.properties`):

    ■ `imq.cluster.brokerlist`

    ■ `imq.cluster.masterbroker` (if necessary)

    ■ `imq.cluster.transport` (if you are using a secure `cluster` connection service)

    When the newly added broker starts, it connects and exchanges data with all the other brokers in the `imq.cluster.brokerlist` value.

2. Modify the `imq.cluster.brokerlist` property of other brokers in the cluster to include the new broker.

    This step is not strictly necessary to add a broker to a functioning cluster. However, should any broker need to be restarted, its `imq.cluster.brokerlist` value must include all other brokers in the cluster, including the newly added broker.

3. Start the new broker.

    If you did not perform step 1, use the `-D` option on the `imqbrokerd` command line to set the property values listed there.

Removing Brokers From a Conventional Cluster

The method you use to remove a broker from a conventional cluster depends on whether you originally created the cluster using a cluster configuration file or by means of command line options.

> **Note:** Before you remove from a conventional cluster the broker instance serving as the cluster's master broker, first change the master broker to another broker instance in the cluster, as described in Changing the Master Broker in a Conventional Cluster with Master Broker

To Remove a Broker From a Conventional Cluster Using a Cluster Configuration File If you originally created a cluster by specifying its member brokers with the `imq.cluster.brokerlist` property in a central cluster configuration file, it isn't necessary to stop the brokers in order to remove one of them. Instead, you can simply edit the configuration file to exclude the broker you want to remove, force the remaining cluster members to reload the cluster configuration, and reconfigure the excluded broker so that it no longer points to the same cluster configuration file:

1. If you are permanently removing the broker from the cluster, prepare it for removal:

   a. Quiesce the broker by using the `imqcmd quiesce bkr` command.

   b. Stop all producer clients connected to the broker.

   c. Drain all messages by waiting for connected consumer clients to consume existing messages.

   Use the `imqcmd query bkr` command periodically to check the number of messages in the broker.

   d. Roll back or commit any prepared open transactions.

   Use the `imqcmd list txn` command to view prepared open transactions, and use the `imqcmd rollback txn` and `imqcmd commit txn` to roll back and commit transactions.

2. Edit the cluster configuration file to remove the excluded broker from the list specified for the `imq.cluster.brokerlist` property.

3. Issue the following command to each broker remaining in the cluster:

   ```
   imqcmd reload cls
   ```
   This forces the brokers to reload the cluster configuration.

4. Stop the broker you're removing from the cluster.

5. Edit that broker's instance configuration file (`config.properties`), removing or specifying a different value for its `imq.cluster.url` property.

To Remove a Broker From a Conventional Cluster Using the Command Line If you used the `imqbrokerd` command from the command line to connect the brokers into a cluster, you must stop each of the brokers and then restart them, specifying the new set of cluster members on the command line:

1. If you are permanently removing the broker from the cluster, prepare it for removal:

   a. Quiesce the broker by using the `imqcmd quiesce bkr` command.

b. Stop all producer clients connected to the broker.

c. Drain all messages by waiting for connected consumer clients to consume existing messages.

Use the `imqcmd query bkr` command periodically to check the number of messages in the broker.

d. Roll back or commit any prepared open transactions.

Use the `imqcmd list txn` command to view prepared open transactions, and use the `imqcmd rollback txn` and `imqcmd commit txn` to roll back and commit transactions.

2. Stop each broker in the cluster, using the `imqcmd` command.

3. Restart the brokers that will remain in the cluster, using the `imqbrokerd` command's `-cluster` option to specify only those remaining brokers.

For example, suppose you originally created a cluster consisting of brokers *A, B,* and *C* by starting each of the three with the command

```
imqbrokerd  -cluster A,B,C
```
To remove broker *A* from the cluster, restart brokers *B* and *C* with the command

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

Changing the Master Broker in a Conventional Cluster with Master Broker

Message Queue provides two ways to change the broker instance serving as the master broker to a different broker instance in the cluster:

- Dynamically while the cluster is running

- Manually by stopping the cluster and migrating the configuration change record from one broker to another

To change the master broker dynamically, you must first configure the brokers in the cluster to support dynamic changing of the master broker.

To Configure a Cluster to Support Dynamic Changing of the Master Broker

1. In the properties file for each broker of the cluster, set the `imq.cluster.dynamicChangeMasterBrokerEnabled` property to `true`.

If using a cluster configuration file, you can instead set the `imq.cluster.dynamicChangeMasterBrokerEnabled` property to `true` in the cluster configuration file.

2. In the properties file for each broker of the cluster, set the `imq.cluster.masterbroker` property to the initial master broker.

When the `imq.cluster.dynamicChangeMasterBrokerEnabled` property is set to `true`, the `imq.cluster.masterbroker` property cannot be specified on the command line to start a broker. Therefore, it must be set in the brokers' properties files, or in the cluster configuration file if one is being used.

To Change the Master Broker Dynamically While the Cluster Is Running To dynamically change the broker instance serving as the master broker to a different broker instance in the cluster, use the `imqcmd changemaster cls` command.

Follow this procedure, for example, before you remove from a cluster the broker instance serving as the master broker.

> **Caution:** Do not use the `imqcmd changemaster cls` command to dynamically change the master broker in a Message Queue cluster managed by GlassFish Server as an Embedded or Local JMS host. Instead, use the `asadmin change-master-broker` command as described in "To Change the Master Broker in an Embedded or Local Broker Cluster" in *GlassFish Server Open Source Edition High Availability Administration Guide*.

Before You Begin

To ensure a successful dynamic changing of the master broker, verify that all brokers in the cluster are running before issuing the `imqcmd changemaster cls` command.

1. On the current master broker, run the `imqcmd changemaster cls` command, using the `-o` to specify the new master broker:

   ```
   imqcmd changemaster cls -o imq.cluster.masterbroker=newMaster
   ```

   The value *newMaster* has the form *hostName*:*portNumber*, where *hostName* and *portNumber* are the Port Mapper host name and port number, respectively, of the new master broker's host.

   The broker returns one of the following status values for the operation:

   OK
   The operation was successful. The new master broker is now the master broker for the cluster, and the old master broker is now a normal broker in the cluster. If any other brokers in the cluster were unreachable and so could not be notified of the change, they must be restarted after manually updating their configurations to refer to the new master broker.

   BAD_REQUEST, NOT_ALLOWED, UNAVAILABLE or PRECONDITION_FAILED
   The operation failed, and the cluster's configuration was unchanged. The old master broker is still the master broker for the cluster.

   Any other value
   The operation failed. Use the `imqcmd query bkr` command on the old master broker to discover which broker is the master broker:

   - If the master broker listed is the old master broker, the failure occurred before the cluster's configuration change record was transferred to the new master broker. In this case, retry the command.

   - If the master broker listed is the new master broker, the cluster's configuration change record was transferred successfully to the new master broker, but some other activity failed later in the operation. In this case, stop all brokers in the cluster, manually update their configurations to refer to the new master broker, and then restart them all.

To Change the Master Broker Manually

1. Stop all brokers in the cluster.

2. Save the configuration change record in the old master broker by using the `-backup` option of the `imqbrokerd` command:

   ```
   imqbrokerd -backup backupFile
   ```

3. Stop the old master broker after the configuration change record has been saved.

4. Update the `imq.cluster.masterbroker` property to the new master broker in the configurations for all brokers in the cluster.

   Additionally, if necessary, update the `imq.cluster.brokerlist` property in the configurations for all brokers in the cluster.

5. Start the new master broker, restoring the saved configuration change record by using the `-restore` option:

   ```
   imqbrokerd -restore backupFile
   ```

   When using this command, specify as *backupFile* the file you saved in Step 1.

6. Start the other brokers in the cluster.

Managing a Conventional Cluster's Configuration Change Record

As noted earlier, a conventional cluster maintains a configuration change record to keep track of any changes in the cluster's persistent state. This configuration change record is maintained either by the master broker or in a shared JDBC data store, depending on the type of the conventional cluster.

Because of the important information that the configuration change record contains, it is important to back it up regularly so that it can be restored in case of failure. Although restoring from a backup will lose any changes in the cluster's persistent state that have occurred since the backup was made, frequent backups can minimize this potential loss of information. The backup and restore operations also have the positive effect of compressing and optimizing the change history contained in the configuration change record, which can grow significantly over time.

To Back Up the Configuration Change Record in a Master Broker

1. Use the `-backup` option of the `imqbrokerd` command, specifying the name of the backup file.

   For example:

   ```
   imqbrokerd  -backup mybackuplog
   ```

To Back Up the Configuration Change Record in a Shared JDBC Data Store

1. Use the `imqdbmgr backup sharecc_tbl` command to back up the configuration change record:

   ```
   imqdbmgr backup sharecc_tbl -file fileName -Dimq.cluster.url=clusterConfigUrl
   ```

To Restore the Configuration Change Record to a Master Broker

1. Shut down all brokers in the cluster.

2. Restore the master broker's configuration change record from the backup file.

   The command is

   ```
   imqbrokerd  -restore mybackuplog
   ```

3. If you assign a new name or port number to the master broker, update the `imq.cluster.brokerlist` and `imq.cluster.masterbroker` properties accordingly in the cluster configuration file.

4. Restart all brokers in the cluster.

To Restore the Configuration Change Record to a Shared JDBC Data Store

1. Shut down all brokers in the cluster.

2. Use the `imqdbmgr recreate sharecc_tbl` command to delete the existing configuration change record and then re-create the table:

    ```
    imqdbmgr recreate sharecc_tbl -Dimq.cluster.url=clusterConfigUrl
    ```

3. Use the `imqdbmgr restore sharecc_tbl` command to restore the configuration change record:

    ```
    imqdbmgr restore sharecc_tbl -file fileName -Dimq.cluster.url=clusterConfigUrl
    ```

4. Start all the brokers in the cluster.

Converting Between Types of Conventional Clusters

To convert between types of conventional clusters, you change where the configuration change record is maintained: in a master broker or in a shared JDBC data store. The following topics provide instructions to convert between types:

- To Convert from Using a Master Broker to Using a Shared JDBC Data Store
- To Convert from Using a Shared JDBC Data Store to Using a Master Broker

To Convert from Using a Master Broker to Using a Shared JDBC Data Store

1. Shut down all brokers in the cluster.

2. Back up the configuration change record in the master broker as described in To Back Up the Configuration Change Record in a Master Broker.

3. Shut down the master broker.

4. Edit the cluster configuration file, configuring the cluster as a conventional cluster of peer brokers:

    - Set the `imq.cluster.nomasterbroker` property to `true`.
    - Set additional properties as described in Additional Properties for Conventional Clusters of Peer Brokers.

5. Using the backup file saved in Step 2, restore the configuration change record to the shared JDBC data store as described in To Restore the Configuration Change Record to a Shared JDBC Data Store.

6. Start all the brokers in the cluster.

To Convert from Using a Shared JDBC Data Store to Using a Master Broker

1. Shut down all brokers in the cluster.

2. Back up the configuration change record in the shared JDBC data store as described in To Back Up the Configuration Change Record in a Shared JDBC Data Store.

3. Edit the cluster configuration file, configuring the cluster as a conventional cluster with master broker:

    - Set the `imq.cluster.nomasterbroker` property to `false`.
    - Set additional properties as described in Additional Properties for Conventional Clusters with Master Broker.

4. Using the backup file saved in Step 2, restore the configuration change record to the master broker as described in To Restore the Configuration Change Record to a Master Broker.

5. Start all the brokers in the cluster.

Managing Enhanced Clusters

This section presents step-by-step procedures for performing a variety of administrative tasks for an enhanced cluster:

- Connecting Brokers into an Enhanced Cluster

- Adding and Removing Brokers in an Enhanced Cluster

- Restarting a Failed Broker

- Preventing or Forcing Broker Failover

- Backing up a Shared Data Store

Connecting Brokers into an Enhanced Cluster

Because enhanced clusters are self-configuring, there is no need to explicitly specify the list of brokers to be included in the cluster. Instead, all that is needed is to set each broker's configuration properties appropriately and then start the broker; as long as its properties are set properly, it will automatically be incorporated into the cluster. Enhanced Broker Cluster Properties describes the required properties, which include vendor-specific JDBC database properties.

> **Note:** In addition to creating an enhanced cluster as described in this section, you must also configure clients to successfully reconnect to a failover broker in the event of broker or connection failure. You do this by setting the `imqReconnectAttempts` connection factory attribute to a value of `-1`.

The property values needed for brokers in an enhanced cluster can be set separately in each broker's instance configuration file, or they can be specified in a cluster configuration file that all the brokers reference. The procedures are as follows:

To Connect Brokers Using a Cluster Configuration File The method best suited for production systems is to use a cluster configuration file to specify the configuration of the cluster.

1. Create a cluster configuration file specifying the cluster's high-availability-related configuration properties.

   Enhanced Broker Cluster Properties shows the required property values. However, do *not* include the `imq.brokerid` property in the cluster configuration file; this must be specified separately for each individual broker in the cluster.

2. Specify any additional, vendor-specific JDBC configuration properties that might be needed.

   The vendor-specific properties required for MySQL are shown in Example 8–1.

3. For each broker in the cluster:

   a. Start the broker at least once, using the `imqbrokerd` command.

The first time a broker instance is run, an instance configuration file (`config.properties`) is automatically created.

b. Shut down the broker.

Use the `imqcmd shutdown bkr` command.

c. Edit the instance configuration file to specify the location of the cluster configuration file.

In the broker's instance configuration file, set the `imq.cluster.url` property to point to the location of the cluster configuration file you created in step 1.

d. Specify the broker identifier.

Set the `imq.brokerid` property in the instance configuration file to the broker's unique broker identifier. This value must be different for each broker.

4. Place a copy of, or a symbolic link to, your JDBC driver's `.jar` file in `IMQ_HOME/lib/ext`, the Message Queue external resource files directory.

5. Create the database tables needed for Message Queue persistence.

Use the `imqdbmgr create tbl` command; see Database Manager Utility.

6. Restart each broker with the `imqbrokerd` command.

The brokers will automatically register themselves into the cluster on startup.

To Connect Brokers Using Instance Configuration Files

1. For each broker in the cluster:

a. Start the broker at least once, using the `imqbrokerd` command.

The first time a broker instance is run, an instance configuration file (`config.properties`) is automatically created.

b. Shut down the broker.

Use the `imqcmd shutdown bkr` command.

c. Edit the instance configuration file to specify the broker's high-availability-related configuration properties.

Enhanced Broker Cluster Properties shows the required property values. Be sure to set the `brokerid` property uniquely for each broker.

d. Specify any additional, vendor-specific JDBC configuration properties that might be needed.

The vendor-specific properties required for MySQL are shown in Example 8–1.

2. Place a copy of, or a symbolic link to, your JDBC driver's `.jar` file in `IMQ_HOME/lib/ext`, the Message Queue external resource files directory.

3. Create the database tables needed for Message Queue persistence.

Use the `imqdbmgr create tbl` command; see Database Manager Utility.

4. Restart each broker with the `imqbrokerd` command.

The brokers will automatically register themselves into the cluster on startup.

Adding and Removing Brokers in an Enhanced Cluster

Because enhanced clusters are self-configuring, the procedures for adding and removing brokers are simpler than for a conventional cluster.

To Add a New Broker to an Enhanced Cluster

1. Set the new broker's high-availability-related properties, as described in the preceding section.

    You can do this either by specifying the individual properties in the broker's instance configuration file (`config.properties`) or, if there is a cluster configuration file, by setting the broker's `imq.cluster.url` property to point to it.

2. Start the new broker with the `imqbrokerd` command.

    The broker will automatically register itself into the cluster on startup.

To Remove a Broker from an Enhanced Cluster

1. Make sure the broker is not running.

    If necessary, use the command

    ```
    imqcmd shutdown bkr
    ```
    to shut down the broker.

2. Remove the broker from the cluster with the command

    ```
    imqdbmgr remove bkr
    ```
    This command deletes all database tables for the corresponding broker.

Restarting a Failed Broker

After a broker has failed, you can restart it using the `imqbrokerd` command. Normally, the broker will automatically be re-registered into the cluster on startup.

However, if the broker slated to take over the failed broker's persistent data failed as it was taking over the persistent data, the running brokers in the cluster will not permit the failed broker to rejoin the cluster for 60 seconds or twice the value of `imq.cluster.monitor.interval` in seconds, whichever is greater.

Preventing or Forcing Broker Failover

Although the takeover of a failed broker's persistent data by a failover broker in an enhanced cluster is normally automatic, there may be times when you want to prevent such failover from occurring. To suppress automatic failover when shutting down a broker, use the `-nofailover` option to the `imqcmd shutdown bkr` subcommand:

```
imqcmd shutdown bkr  -nofailover   -b hostName:portNumber
```
where *hostName* and *portNumber* are the host name and port number of the broker to be shut down.

Conversely, you may sometimes need to force a broker failover to occur manually. (This might be necessary, for instance, if a failover broker were to itself fail before completing the takeover process.) In such cases, you can initiate a failover manually from the command line: first shut down the broker to be taken over with the `-nofailover` option, as shown above, then issue the command

```
imqcmd takeover bkr  -n brokerID
```
where *brokerID* is the broker identifier of the broker to be taken over. If the specified broker appears to be running, the Command utility will display a confirmation message:

The broker associated with *brokerID*last accessed the database #seconds ago.
Do you want to take over for this broker?

You can suppress this message, and force the takeover to occur unconditionally, by using the -f option to the imqcmd takeover bkr command:

```
imqcmd takeover bkr  -f  -n brokerID
```

> **Note:** The imqcmd takeover bkr subcommand is intended only for use in failed-takeover situations. You should use it only as a last resort, and not as a general way of forcibly taking over a running broker.

Backing up a Shared Data Store

For durability and reliability, it is a good idea to back up an enhanced cluster's shared data store periodically to backup files. This creates a snapshot of the data store that you can then use to restore the data in case of catastrophic failure. The command for backing up the data store is

```
imqdbmgr backup  -dir backupDir
```

where *backupDir* is the path to the directory in which to place the backup files. To restore the data store from these files, use the command

```
imqdbmgr restore  -dir backupDir
```

Before restoring the data store, you should shut down all brokers in the enhanced cluster.

Converting a Conventional Cluster to an Enhanced Cluster

The best approach to converting a conventional broker cluster to an enhanced broker cluster is to drain your messaging system of all persistent data before attempting the conversion. This lets you create a new shared data store without worrying about loss of data. However, if you are using individual JDBC-based data stores for your brokers, a utility is available for converting a standalone datastore to a shared data store.

Cluster Conversion : File-Based Data Store

If the brokers in your conventional cluster are using file-based data stores, use the following procedure to convert to an enhanced cluster.

1. Drain down your messaging system of all persistent data.

    Stop all producer clients from producing messages, and wait for all messages in the system to be consumed.

2. Shut down all client applications.

3. Shut down all brokers in the conventional cluster.

4. Reconfigure all brokers for an enhanced cluster.

    See Enhanced Broker Cluster Properties. It is recommended that you use a cluster configuration file to specify cluster configuration property values, such as the imq.cluster.clusterid, imq.persist.store, and additional shared JDBC database properties.

5. Start all brokers in the enhanced cluster.

    See Connecting Brokers into an Enhanced Cluster.

6. Configure client applications to re-connect to failover brokers.

Client re-connection behavior is specified by connection handling attributes of the connection factory administered objects (see the Connection Handling). In the case of enhanced broker clusters, the `imqAddressList`, `imqAddressListBehavior`, and `imqAddressListIterations` attributes are ignored, however the `imqReconnectAttempts` attribute should be set to a value of `-1` (unlimited).

7. Start all client applications.

8. Resume messaging operations

Cluster Conversion: JDBC-Based Data Store

If the brokers in your conventional cluster are using JDBC-based data stores, use the following procedure to convert to an enhanced cluster. The procedure assumes that individual standalone broker data stores reside on the same JDBC database server.

1. Back up all persistent data in the standalone JDBC-based data store of each broker.

   Use proprietary JDBC database tools.

2. Shut down all client applications.

3. Shut down all brokers in the conventional cluster.

4. Convert each standalone data store to a shared data store.

   Use the Message Queue Database Manager utility (`imqdbmgr`) subcommand

   ```
   imqdbmgr upgrade hastore
   ```
   to convert an existing standalone JDBC database to a shared JDBC database.

5. Reconfigure all brokers for an enhanced cluster.

   See Enhanced Broker Cluster Properties. It is recommended that you use a cluster configuration file to specify cluster configuration property values, such as the `imq.cluster.clusterid`, `imq.persist.store`, and additional shared JDBC database properties.

6. Start all brokers in the enhanced cluster.

   See Connecting Brokers into an Enhanced Cluster.

7. Configure client applications to re-connect to failover brokers.

   Client re-connection behavior is specified by connection handling attributes of the connection factory administered objects (see the Connection Handling). In the case of enhanced broker clusters, the `imqAddressList`, `imqAddressListBehavior`, and `imqAddressListIterations` attributes are ignored, however the `imqReconnectAttempts` attribute should be set to a value of `-1` (unlimited).

8. Start all client applications.

9. Resume messaging operations.

11

Managing Administered Objects

Administered objects encapsulate provider-specific configuration and naming information, enabling the development of client applications that are portable from one JMS provider to another. A Message Queue administrator typically creates administered objects for client applications to use in obtaining broker connections for sending and receiving messages.

This chapter tells how to use the Object Manager utility (`imqobjmgr`) to create and manage administered objects. It contains the following sections:

- Object Stores
- Administered Object Attributes
- Using the Object Manager Utility

Object Stores

Administered objects are placed in a readily available object store where they can be accessed by client applications by means of the Java Naming and Directory Interface (JNDI). There are two types of object store you can use: a standard Lightweight Directory Access Protocol (LDAP) directory server or a directory in the local file system.

LDAP Server Object Stores

An LDAP server is the recommended object store for production messaging systems. LDAP servers are designed for use in distributed systems and provide security features that are useful in production environments.

LDAP implementations are available from a number of vendors. To manage an object store on an LDAP server with Message Queue administration tools, you may first need to configure the server to store Java objects and perform JNDI lookups; see the documentation provided with your LDAP implementation for details.

To use an LDAP server as your object store, you must specify the attributes shown in Table 11–1. These attributes fall into the following categories:

- **Initial context.** The `java.naming.factory.initial` attribute specifies the initial context for JNDI lookups on the server. The value of this attribute is fixed for a given LDAP object store.

- **Location.** The `java.naming.provider.url` attribute specifies the URL and directory path for the LDAP server. You must verify that the specified directory path exists.

- **Security.** The `java.naming.security.principal`, `java.naming.security.credentials`, and `java.naming.security.authentication` attributes govern the authentication of callers attempting to access the object store. The exact format and values of these attributes depend on the LDAP service provider; see the documentation provided with your LDAP implementation for details and to determine whether security information is required on all operations or only on those that change the stored data.

Table 11–1   LDAP Object Store Attributes

Attribute	Description
`java.naming.factory.initial`	Initial context for JNDI lookup **Example:** - `com.sun.jndi.ldap.LdapCtxFactory`
`java.naming.provider.url`	Server URL and directory path **Example:** - `ldap://myD.com:389/ou=mq1,o=App` where administered objects are stored in the directory `/App/mq1`.
`java.naming.security.principal`	Identity of the principal for authenticating callers The format of this attribute depends on the authentication scheme: for example, - `uid=homerSimpson,ou=People,o=mq` If this attribute is unspecified, the behavior is determined by the LDAP service provider.
`java.naming.security.credentials`	Credentials of the authentication principal The value of this attribute depends on the authentication scheme: for example, it might be a hashed password, a clear-text password, a key, or a certificate. If this property is unspecified, the behavior is determined by the LDAP service provider.
`java.naming.security.authentication`	Security level for authentication: - `none`: No security - `simple`: Simple security - `strong`: Strong security For example, if you specify `simple`, you will be prompted for any missing principal or credential values. This will allow you a more secure way of providing identifying information. If this property is unspecified, the behavior is determined by the LDAP service provider.

File-System Object Stores

Message Queue also supports the use of a directory in the local file system as an object store for administered objects. While this approach is not recommended for production systems, it has the advantage of being very easy to use in development environments. Note, however, that for a directory to be used as a centralized object store for clients deployed across multiple computer nodes, all of those clients must have access to the directory. In addition, any user with access to the directory can use Message Queue administration tools to create and manage administered objects.

To use a file-system directory as your object store, you must specify the attributes shown in Table 11–2. These attributes have the same general meanings described above for LDAP object stores; in particular, the `java.naming.provider.url` attribute specifies the directory path of the directory holding the object store. This directory must exist and have the proper access permissions for the user of Message Queue administration tools as well as the users of the client applications that will access the store.

Table 11–2   File-system Object Store Attributes

Attribute	Description
`java.naming.factory.initial`	Initial context for JNDI lookup **Example:** ■ `com.sun.jndi.fscontext.RefFSContextFactory`
`java.naming.provider.url`	Directory path **Example:** ■ `file:///C:/myapp/mqobjs`

Administered Object Attributes

Message Queue administered objects are of two basic kinds:

■ Connection factories are used by client applications to create connections to brokers.

■ Destinations represent locations on a broker with which client applications can exchange (send and retrieve) messages.

Each type of administered object has certain attributes that determine the object's properties and behavior. This section describes how to use the Object Manager command line utility (`imqobjmgr`) to set these attributes; you can also set them with the GUI Administration Console, as described in Working With Administered Objects.

Connection Factory Attributes

Client applications use connection factory administered objects to create connections with which to exchange messages with a broker. A connection factory's attributes define the properties of all connections it creates. Once a connection has been created, its properties cannot be changed; thus the only way to configure a connection's properties is by setting the attributes of the connection factory used to create it.

Message Queue defines two classes of connection factory objects:

■ `ConnectionFactory` objects support normal messaging and nondistributed transactions.

■ `XAConnectionFactory` objects support distributed transactions.

Both classes share the same configuration attributes, which you can use to optimize resources, performance, and message throughput. These attributes are listed and described in detail in Administered Object Attribute Reference and are discussed in the following sections below:

■ Connection Handling

■ Client Identification

■ Reliability And Flow Control

- [Queue Browser and Server Sessions](#)
- [Standard Message Properties](#)
- [Message Header Overrides](#)

Connection Handling

Connection handling attributes specify the broker address to which to connect and, if required, how to detect connection failure and attempt reconnection. They are summarized in Table 19–1.

Broker Address List The most important connection handling attribute is `imqAddressList`, which specifies the broker or brokers to which to establish a connection. The value of this attribute is a string containing a broker address or (in the case of a broker cluster) multiple addresses separated by commas. Broker addresses can use a variety of addressing schemes, depending on the connection service to be used (see Configuring Connection Services) and the method of establishing a connection:

- `mq` uses the broker's Port Mapper to assign a port dynamically for either the `jms` or `ssljms` connection service.

- `mqtcp` bypasses the Port Mapper and connects directly to a specified port, using the `jms` connection service.

- `mqssl` makes a Secure Socket Layer (SSL) connection to a specified port, using the `ssljms` connection service.

- `http` makes a Hypertext Transport Protocol (HTTP) connection to a Message Queue tunnel servlet at a specified URL, using the `httpjms` connection service.

- `https` makes a Secure Hypertext Transport Protocol (HTTPS) connection to a Message Queue tunnel servlet at a specified URL, using the `httpsjms` connection service.

These addressing schemes are summarized in Table 19–2.

The general format for each broker address is

`scheme:// address`
where *scheme* is one of the addressing schemes listed above and *address* denotes the broker address itself. The exact syntax for specifying the address varies depending on the addressing scheme, as shown in the "Description" column of Table 19–2. Table 19–3 shows examples of the various address formats.

In a multiple-broker cluster environment, the address list can contain more than one broker address. If the first connection attempt fails, the Message Queue client runtime will attempt to connect to another address in the list, and so on until the list is exhausted. Two additional connection factory attributes control the way this is done:

- `imqAddressListBehavior` specifies the order in which to try the specified addresses. If this attribute is set to the string `PRIORITY`, addresses will be tried in the order in which they appear in the address list. If the attribute value is `RANDOM`, the addresses will instead be tried in random order; this is useful, for instance, when many Message Queue clients are sharing the same connection factory object, to prevent them from all attempting to connect to the same broker address.

- `imqAddressListIterations` specifies how many times to cycle through the list before giving up and reporting failure. A value of `-1` denotes an unlimited number of iterations: the client runtime will keep trying until it succeeds in establishing a connection or until the end of time, whichever occurs first.

> **Note:** Because enhanced clusters are self-configuring (see Cluster Configuration Properties and Connecting Brokers into an Enhanced Cluster), their membership can change over time as brokers enter and leave the cluster. In this type of cluster, the value of each member broker's `imqAddressList` attribute is updated dynamically so that it always reflects the cluster's current membership.

Automatic Reconnection By setting certain connection factory attributes, you can configure a client to reconnect automatically to a broker in the event of a failed connection. For standalone brokers or those belonging to a conventional broker cluster (see "Conventional Clusters" in *Open Message Queue Technical Overview*), you enable this behavior by setting the connection factory's `imqReconnectEnabled` attribute to `true`. The `imqReconnectAttempts` attribute controls the number of reconnection attempts to a given broker address; `imqReconnectInterval` specifies the interval, in milliseconds, to wait between attempts.

If the broker is part of a conventional cluster, the failed connection can be restored not only on the original broker, but also on a different one in the cluster. If reconnection to the original broker fails, the client runtime will try the other addresses in the connection factory's broker address list (`imqAddressList`). The `imqAddressListBehavior` and `imqAddressListIterations` attributes control the order in which addresses are tried and the number of iterations through the list, as described in the preceding section. Each address is tried repeatedly at intervals of `imqReconnectInterval` milliseconds, up to the maximum number of attempts specified by `imqReconnectAttempts`.

Note, however, that in a conventional cluster, such automatic reconnection only provides connection failover and not data failover: persistent messages and other state information held by a failed or disconnected broker can be lost when the client is reconnected to a different broker instance. While attempting to reestablish a connection, Message Queue does maintain objects (such as sessions, message consumers, and message producers) provided by the client runtime. Temporary destinations are also maintained for a time when a connection fails, because clients might reconnect and access them again; after giving clients time to reconnect and use these destinations, the broker will delete them. In circumstances where the client-side state cannot be fully restored on the broker on reconnection (for instance, when using transacted sessions, which exist only for the duration of a connection), automatic reconnection will not take place and the connection's exception handler will be called instead. It is then up to the client application to catch the exception, reconnect, and restore state.

By contrast, in an enhanced cluster (see "Enhanced Clusters" in *Open Message Queue Technical Overview*), another broker can take over a failed broker's persistent state and proceed to deliver its pending messages without interruption of service. In this type of cluster, automatic reconnection is always enabled. The connection factory's `imqReconnectEnabled`, `imqAddressList`, and `imqAddressListIterations` attributes are ignored. The client runtime is automatically redirected to the failover broker. Because there might be a short time lag during which the failover broker takes over from the failed broker, the `imqReconnectAttempts` connection factory attribute should be set to a value of `-1` (client runtime continues connect attempts until successful).

Automatic reconnection supports all client acknowledgment modes for message consumption. Once a connection has been reestablished, the broker will redeliver all unacknowledged messages it had previously delivered, marking them with a `Redeliver` flag. Client applications can use this flag to determine whether a message has already been consumed but not yet acknowledged. (In the case of nondurable

subscribers, however, the broker does not hold messages once their connections have been closed. Thus any messages produced for such subscribers while the connection is down cannot be delivered after reconnection and will be lost.) Message production is blocked while automatic reconnection is in progress; message producers cannot send messages to the broker until after the connection has been reestablished.

Periodic Testing (Pinging) of Connections The Message Queue client runtime can be configured to periodically test, or "ping," a connection, allowing connection failures to be detected preemptively before an attempted message transmission fails. Such testing is particularly important for client applications that only consume messages and do not produce them, since such applications cannot otherwise detect when a connection has failed. Clients that produce messages only infrequently can also benefit from this feature.

The connection factory attribute `imqPingInterval` specifies the frequency, in seconds, with which to ping a connection. By default, this interval is set to 30 seconds; a value of `-1` disables the ping operation.

The response to an unsuccessful ping varies from one operating-system platform to another. On some platforms, an exception is immediately thrown to the client application's exception listener. (If the client does not have an exception listener, its next attempt to use the connection will fail.) Other platforms may continue trying to establish a connection to the broker, buffering successive pings until one succeeds or the buffer overflows.

Client Identification

The connection factory attributes listed in support client authentication and the setting of client identifiers for durable subscribers.

Client Authentication All attempts to connect to a broker must be authenticated by user name and password against a user repository maintained by the message service. The connection factory attributes `imqDefaultUsername` and `imqDefaultPassword` specify a default user name and password to be used if the client does not supply them explicitly when creating a connection.

As a convenience for developers who do not wish to bother populating a user repository during application development and testing, Message Queue provides a guest user account with user name and password both equal to `guest`. This is also the default value for the `imqDefaultUsername` and `imqDefaultPassword` attributes, so that if they are not specified explicitly, clients can always obtain a connection under the guest account. In a production environment, access to broker connections should be restricted to users who are explicitly registered in the user repository.

Client Identifier A connection canprovide a unique client identifier whenever the broker must maintain a persistent state on behalf of a client. Message Queue uses such client identifiers to keep track of durable subscribers to a topic destination. When a durable subscriber becomes inactive, the broker retains all incoming messages for the topic and delivers them when the subscriber becomes active again. The broker identifies the subscriber by means of its client identifier.

While it is possible for a client application to set its own client identifier programmatically using the connection object's `setClientID` method, this makes it difficult to coordinate client identifiers to ensure that each is unique. It is generally better to have Message Queue automatically assign a unique identifier when creating a connection on behalf of a client. This can be done by setting the connection factory's `imqConfiguredClientID` attribute to a value of the form

${u}*factoryID*

The characters `${u}` must be the first four characters of the attribute value. (Any character other than `u` between the braces will cause an exception to be thrown on connection creation; in any other position, these characters have no special meaning and will be treated as plain text.) The value for *factoryID* is a character string uniquely associated with this connection factory object.

When creating a connection for a particular client, Message Queue will construct a client identifier by replacing the characters `${u}` with `${u:`*userName*`}`, where *userName* is the user name authenticated for the connection. This ensures that connections created by a given connection factory, although identical in all other respects, will each have their own unique client identifier. For example, if the user name is `Calvin` and the string specified for the connection factory's `imqConfiguredClientID` attribute is `${u}Hobbes`, the client identifier assigned will be `${u:Calvin}Hobbes`.

> **Note:** This scheme will not work if two clients both attempt to obtain connections using the default user name `guest`, since each would have a client identifier with the same `${u}` component. In this case, only the first client to request a connection will get one; the second client's connection attempt will fail, because Message Queue cannot create two connections with the same client identifier.

Even if you specify a client identifier with `imqConfiguredClientID`, client applications can override this setting with the connection method `setClientID`. You can prevent this by setting the connection factory's `imqDisableSetClientID` attribute to `true`. Note that for an application that uses durable subscribers, the client identifier *must* be set one way or the other: either administratively with `imqConfiguredClientID` or programmatically with `setClientID`.

Reliability And Flow Control

Because "payload" messages sent and received by clients and control messages (such as broker acknowledgments) used by Message Queue itself pass over the same client-broker connection, excessive levels of payload traffic can interfere with the delivery of control messages. To help alleviate this problem, the connection factory attributes listed in Table 19–5 allow you to manage the relative flow of the two types of message. These attributes fall into four categories:

- **Acknowledgment timeout** specifies the maximum time (`imqAckTimeout`) to wait for a broker acknowledgment before throwing an exception.

- **Connection flow metering** limits the transmission of payload messages to batches of a specified size (`imqConnectionFlowCount`), ensuring periodic opportunities to deliver any accumulated control messages.

- **Connection flow control** limits the number of payload messages (`imqConnectionFlowLimit`) that can be held pending on a connection, waiting to be consumed. When the limit is reached, delivery of payload messages to the connection is suspended until the number of messages awaiting consumption falls below the limit. Use of this feature is controlled by a boolean flag (`imqConnectionFlowLimitEnabled`).

- **Consumer flow control** limits the number of payload messages (`imqConsumerFlowLimit`) that can be held pending for any single consumer, waiting to be consumed. (This limit can also be specified as a property of a specific queue destination, `consumerFlowLimit`.) When the limit is reached, delivery of

payload messages to the consumer is suspended until the number of messages awaiting consumption, as a percentage of `imqConsumerFlowLimit`, falls below the limit specified by the `imqConsumerFlowThreshold` attribute. This helps improve load balancing among multiple consumers by preventing any one consumer from starving others on the same connection.

When the JMS resource adapter, jmsra, is used to consume messages in a GlassFish Server cluster, this behavior is defined using different properties, as described in About Shared Topic Subscriptions for Clustered Containers.

The use of any of these flow control techniques entails a trade-off between reliability and throughput; see Client Runtime Message Flow Adjustments for further discussion.

Queue Browser and Server Sessions

Table 19–6 lists connection factory attributes affecting client queue browsing and server sessions. The `imqQueueBrowserMaxMessagesPerRetrieve` attribute specifies the maximum number of messages to retrieve at one time when browsing the contents of a queue destination; `imqQueueBrowserRetrieveTimeout` gives the maximum waiting time for retrieving them. (Note that `imqQueueBrowserMaxMessagesPerRetrieve` does not affect the total number of messages browsed, only the way they are batched for delivery to the client runtime: fewer but larger batches or more but smaller ones. Changing the attribute's value may affect performance, but will not affect the total amount of data retrieved; the client application will always receive all messages in the queue.) The boolean attribute `imqLoadMaxToServerSession` governs the behavior of connection consumers in an application server session: if the value of this attribute is `true`, the client will load up to the maximum number of messages into a server session; if `false`, it will load only a single message at a time.

Standard Message Properties

The *Java Message Service Specification* defines certain standard message properties, which JMS providers (such as Message Queue) may optionally choose to support. By convention, the names of all such standard properties begin with the letters `JMSX`. The connection factory attributes listed in Table 19–7 control whether the Message Queue client runtime sets certain of these standard properties. For produced messages, these include the following:

JMSXUserID
Identity of the user sending the message

JMSXAppID
Identity of the application sending the message

JMSXProducerTXID
Transaction identifier of the transaction within which the message was produced

For consumed messages, they include

JMSXConsumerTXID
Transaction identifier of the transaction within which the message was consumed

JMSXRcvTimestamp
Time the message was delivered to the consumer

Message Header Overrides

You can use the connection factory attributes listed in Table 19–8 to override the values set by a client for certain JMS message header fields. The settings you specify will be used for all messages produced by connections obtained from that connection factory. Header fields that you can override in this way are

JMSDeliveryMode
Delivery mode (persistent or nonpersistent)

JMSExpiration
Expiration time

JMSPriority
Priority level

There are two attributes for each of these fields: one boolean, to control whether the field can be overridden, and another to specify its value. For instance, the attributes for setting the priority level are `imqOverrideJMSPriority` and `imqJMSPriority`. There is also an additional attribute, `imqOverrideJMSHeadersToTemporaryDestinations`, that controls whether override values apply to temporary destinations.

Note: Because overriding message headers may interfere with the needs of specific applications, these attributes should only be used in consultation with an application's designers or users.

Destination Attributes

The destination administered object that identifies a physical queue or topic destination has only two attributes, listed in Table 19–9. The important one is `imqDestinationName`, which gives the name of the physical destination that this administered object represents; this is the name that was specified with the `-n` option to the `imqcmd create dst` command that created the physical destination. (Note that there is not necessarily a one-to-one relationship between destination administered objects and the physical destinations they represent: a single physical destination can be referenced by more than one administered object, or by none at all.) There is also an optional descriptive string, `imqDestinationDescription`, which you can use to help identify the destination object and distinguish it from others you may have created.

Using the Object Manager Utility

The Message Queue Object Manager utility (`imqobjmgr`) allows you to create and manage administered objects. The `imqobjmgr` command provides the following subcommands for performing various operations on administered objects:

add
Add an administered object to an object store

delete
Delete an administered object from an object store

list
List existing administered objects in an object store

query
Display information about an administered object

update
Modify the attributes of an administered object

See Object Manager Utility for reference information about the syntax, subcommands, and options of the imqobjmgr command.

Most Object Manager operations require you to specify the following information as options to the imqobjmgr command:

- The **JNDI lookup name** (-l) of the administered object

  This is the logical name by which client applications can look up the administered object in the object store, using the Java Naming and Directory Interface.

- The **attributes of the JNDI object store** (-j)

  See Object Stores for information on the possible attributes and their values.

- The **type** (-t) of the administered object

  Possible types include the following:

  q
  Queue destination

  t
  Topic destination

  cf
  Connection factory

  qf
  Queue connection factory

  tf
  Topic connection factory

  xcf
  Connection factory for distributed transactions

  xqf
  Queue connection factory for distributed transactions

  xtf
  Topic connection factory for distributed transactions

- The **attributes** (-o) of the administered object

  See Administered Object Attributes for information on the possible attributes and their values.

Connecting to a Secured LDAP Server (ldaps)

When using imqobjmgr utility to connect to a secured LDAP server that uses the ldaps protocol, you might first have to install the LDAP server's certificate in the JDK keystore. To perform this task, use the Java SE keytool utility; for example:

```
keytool -import -keystore IMQ_JAVAHOME/lib/security/jssecacerts -alias "ldaps
cert" -file certFile
```

where *certFile* is the certificate file from the LDAP server.

Then, when you run the `imqobjmgr` utility, make sure to specify the same Java installation where the certificate was imported; for example:

```
imqobjmgr -javahome IMQ_JAVAHOME ...
```

Adding Administered Objects

The `imqobjmgr` command's `add` subcommand adds administered objects for connection factories and topic or queue destinations to the object store. Administered objects stored in an LDAP object store must have lookup names beginning with the prefix `cn=`; lookup names in a file-system object store need not begin with any particular prefix, but must not include the slash character (`/`).

> **Note:** The Object Manager lists and displays only Message Queue administered objects. If an object store contains a non-Message Queue 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 create broker connections, add a connection factory administered object for the type of connection to be created: a queue connection factory or a topic connection factory, as the case may be. Example 11–1 shows a command to add a queue connection factory (administered object type `qf`) to an LDAP object store. The object has lookup name `cn=myQCF` and connects to a broker running on host `myHost` at port number `7272`, using the `jms` connection service.

Example 11–1   Adding a Connection Factory

```
imqobjmgr add
   -l "cn=myQCF"
   -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=homerSimpson,ou=People,o=imq"
   -j "java.naming.security.credentials=doh"
   -j "java.naming.security.authentication=simple"
   -t qf
   -o "imqAddressList=mq://myHost:7272/jms"
```

Adding a Destination

When creating an administered object representing a destination, it is good practice to create the physical destination first, before adding the administered object to the object store. Use the Command utility (`imqcmd`) to create the physical destination, as described in Creating and Destroying Physical Destinations.

The command shown in Example 11–2 adds an administered object to an LDAP object store representing a topic destination with lookup name `myTopic` and physical destination name `physTopic`. The command for adding a queue destination would be similar, except that the administered object type (`-t` option) would be `q` (for "queue destination") instead of `t` (for "topic destination").

Example 11–2   Adding a Destination to an LDAP Object Store

```
imqobjmgr add
    -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=homerSimpson,ou=People,o=imq"
    -j "java.naming.security.credentials=doh"
    -j "java.naming.security.authentication=simple"
    -t t
    -o "imqDestinationName=physTopic"
```

Example 11–3 shows the same command, but with the administered object stored in a Solaris file system instead of an LDAP server.

Example 11–3   Adding a Destination to a File-System Object Store

```
imqobjmgr add
    -l "cn=myTopic"
    -j "java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory"
    -j "java.naming.provider.url=file:///home/foo/imq_admin_objects"
    -t t
    -o "imqDestinationName=physTopic"
```

Deleting Administered Objects

To delete an administered object from the object store, use the `imqobjmgr delete` subcommand, specifying the lookup name, type, and location of the object to be deleted. The command shown in Example 11–4 deletes the object that was added in Adding a Destination above.

Example 11–4   Deleting an Administered Object

```
imqobjmgr delete
    -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=homerSimpson,ou=People,o=imq"
    -j "java.naming.security.credentials=doh"
    -j "java.naming.security.authentication=simple"
    -t t
```

Listing Administered Objects

You can use the `imqobjmgr list` subcommand to get a list of all administered objects in an object store or those of a specific type. Example 11–5 shows how to list all administered objects on an LDAP server.

Example 11–5   Listing All Administered 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=homerSimpson,ou=People,o=imq"
-j "java.naming.security.credentials=doh"
-j "java.naming.security.authentication=simple"
```

Example 11–6 lists all queue destinations (type q).

Example 11–6   Listing Administered Objects of a Specific Type

```
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=homerSimpson,ou=People,o=imq"
  -j "java.naming.security.credentials=doh"
  -j "java.naming.security.authentication=simple"
  -t q
```

Viewing Administered Object Information

The imqobjmgr query subcommand displays information about a specified administered object, identified by its lookup name and the attributes of the object store containing it. Example 11–7 displays information about an object whose lookup name is cn=myTopic.

Example 11–7   Viewing Administered Object Information

```
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=homerSimpson,ou=People,o=imq"
  -j "java.naming.security.credentials=doh"
  -j "java.naming.security.authentication=simple"
```

Modifying Administered Object Attributes

To modify the attributes of an administered object, use the imqobjmgr update subcommand. You supply the object's lookup name and location, and use the -o option to specify the new attribute values.

Example 11–8 changes the value of the imqReconnectAttempts attribute for the queue connection factory that was added to the object store in Example 11–1.

Example 11–8   Modifying an Administered Object's Attributes

```
imqobjmgr update
  -l "cn=myQCF"
  -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=homerSimpson,ou=People,o=imq"
  -j "java.naming.security.credentials=doh"
  -j "java.naming.security.authentication=simple"
```

```
                    -t qf
                    -o "imqReconnectAttempts=3"
```

Using Command Files

The -i option to the imqobjmgr command allows you to specify the name of a command file that uses Java property file syntax to represent all or part of the subcommand clause. This feature is especially useful for specifying object store attributes, which typically require a lot of typing and are likely to be the same across multiple invocations of imqobjmgr. Using a command file can also allow you to avoid exceeding the maximum number of characters allowed for the command line.

Example 11–9 shows the general syntax for an Object Manager command file. Note that the version property is not a command line option: it refers to the version of the command file itself (not that of the Message Queue product) and must be set to the value 2.0.

Example 11–9   Object Manager Command File Syntax

```
version=2.0
cmdtype=[ add | delete | list | query | update ]
obj.lookupName=lookup name
objstore.attrs.objStoreAttrName1=value1
objstore.attrs.objStoreAttrName2=value2
    . . .
objstore.attrs.objStoreAttrNameN=valueN
obj.type=[ q | t | cf | qf | tf | xcf | xqf | xtf | e ]
obj.attrs.objAttrName1=value1
obj.attrs.objAttrName2=value2
    . . .
obj.attrs.objAttrNameN=valueN
```

As an example, consider the Object Manager command shown earlier in Example 11–1, which adds a queue connection factory to an LDAP object store. This command can be encapsulated in a command file as shown in Example 11–10. If the command file is named MyCmdFile, you can then execute the command with the command line

```
imqobjmgr  -i MyCmdFile
```

Example 11–10   Example Command File

```
version=2.0
cmdtype=add
obj.lookupName=cn=myQCF
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=homerSimpson,ou=People,o=imq
objstore.attrs.java.naming.security.credentials=doh
objstore.attrs.java.naming.security.authentication=simple
obj.type=qf
obj.attrs.imqAddressList=mq://myHost:7272/jms
```

A command file can also be used to specify only part of the imqobjmgr subcommand clause, with the remainder supplied directly on the command line. For example, the

command file shown in Example 11–11 specifies only the attribute values for an LDAP object store.

Example 11–11   Partial Command File

```
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=homerSimpson,ou=People,o=imq
objstore.attrs.java.naming.security.credentials=doh
objstore.attrs.java.naming.security.authentication=simple
```

You could then use this command file to specify the object store in an `imqobjmgr` command while supplying the remaining options explicitly, as shown in Example 11–12.

Example 11–12   Using a Partial Command File

```
imqobjmgr add
   -l "cn=myQCF"
   -i MyCmdFile
   -t qf
   -o "imqAddressList=mq://myHost:7272/jms"
```

Additional examples of command files can be found in the `IMQ_HOME/examples/imqobjmgr` directory.

12

Configuring and Managing Bridge Services

Message-Oriented Middleware (MOM) systems use a broad spectrum of technologies and standards to provide messaging services. Often, these technologies and standards are incompatible, leading to MOM systems that cannot communicate with each other in a larger enterprise application context.

To alleviate this inability to communicate, Message Queue incorporates the Bridge Service Manager, which supports individual bridge services of various types. Each type of bridge service provides connectivity at the broker level to a MOM technology or standard that would otherwise be unavailable in Message Queue.

This chapter provides information about the administrative components of the Bridge Service Manager, and shows how to configure and manage the two types of bridge services currently available:

- The Bridge Service Manager
- Configuring and Managing JMS Bridge Services
- Configuring and Managing STOMP Bridge Services

The Bridge Service Manager

The Message Queue Bridge Service Manager is an application that runs in same JVM as a broker to manage the bridges configured for the broker. Two administrative components control operation of the Bridge Service Manager:

- Bridge-related broker properties
- The Bridge Manager utility (`imqbridgermgr`)

The following sections introduce these two components.

Bridge-Related Broker Properties

The operation of the Bridge Service Manager is in part controlled by several broker properties. These broker properties, all of which begin with `imq.bridge`, are listed in tables under Bridge Properties. Some of the properties apply to all bridges configured for the broker, while others apply only to a specific bridge. The properties that apply to a specific bridge all begin with `imq.bridge.`*bridgeName*, where *bridgeName* is:

- The same as the type of the bridge for bridge services that support only one bridge instance per broker, such as the STOMP bridge service
- A name you specify for a bridge instance for bridge services that support multiple bridge instances per broker, such as the JMS bridge service

Of all the bridge-related broker properties, the two most important are
`imq.bridge.enabled` and `imq.bridge.activelist`:

- The `imq.bridge.enabled` property controls whether the Bridge Service Manager is
  enabled on the broker.

- The `imq.bridge.activelist` property contains a comma-separated list bridges (by
  name) to be loaded when the broker starts.

To Enable the Bridge Service Manager

1. Set the `imq.bridge.enabled` broker property to `true`.

2. Set the `imq.bridge.admin.user` broker property to the user name of the admin
   user.

3. Set the `imq.bridge.admin.password` broker property to the password of the admin
   user.

   Alternatively, you can specify the password using the `-passfile` option when you
   use the `imqbrokerd` command to start the broker hosting the bridge service
   manager.

4. Set the `imq.bridge.activelist` broker property to a comma-separated list of
   bridges to instantiate at broker startup.

Bridge Manager Utility

The Bridge Manager utility (`imqbridgemgr`) is the interface to the bridge management
functions of the Bridge Service Manager. It provides commands to:

- Stop and start bridges

- Pause and resume bridges

- List configured bridges

- Manage type-dependent subcomponents of bridges, such as the links within a JMS
  bridge service

The `imqbridgemgr` utility uses the same Command Line Syntax as the other Message
Queue utilities:

```
imqbridgemgr subcommand commandArgument [ options ]
```

For example, the following command lists all bridges of type JMS on the broker
localhost:7373:

```
imqbridgemgr list bridge -t jms -b localhost:7373
```

For the complete set of subcommands, command arguments, and options supported
by the `imqbridgemgr` utility, see Bridge Manager Utility.

Logging of Bridge Services

Each bridge managed by the Bridge Service Manager for a broker has its own log file
stored in the `IMQ_VARHOME/instances/`*broker-name*`/bridges/`*bridge-name*`/` directory.

The JMS and STOMP bridge services use the Java logging facility, which can be
configured by the Java logging configuration file. The logging level for a bridge can be
controlled by setting the `imq.bridge.`*bridge-name*`.level` property in the Java logging
configuration file. Then, the Java system property `java.util.logging.config.file`
can be set to the Java logging configuration file when the broker is started; as in:

```
imqbrokerd -Djava.util.logging.config.file=config-file
```

Configuring and Managing JMS Bridge Services

Because the JMS specification does not dictate the communication protocol between brokers and clients, each JMS provider (including Message Queue) has defined and uses its own propriety protocol. This situation has led to non-interoperability across JMS providers.

The JMS bridge service in Message Queue 5.0 closes this gap by enabling a Message Queue broker to map its destinations to destinations in external JMS providers. This mapping effectively allows the Message Queue broker to communicate with clients of the external JMS provider.

The JMS bridge service supports mapping destinations to external JMS providers that:

- Are JMS 1.1 compliant

- Support JNDI administrative objects

- Use connection factories of type `javax.jms.ConnectionFactory` or `javax.jms.XAConnectionFactory`

- Support the XA interfaces as a resource manager for transacted mapping

As an administrative and management convenience, the JMS bridge service supports the creation of any number of JMS bridges in a broker. Each JMS bridge in the broker is identified by a unique name, has its own configuration, and is managed separately from other JMS bridges in the broker.

The following subsections provide information about JMS bridges and how to configure and manage them:

- JMS Bridge Components

- JMS Bridge Features

- Message Processing Sequence Across a Link in a JMS Bridge

- Configuring a JMS Bridge

- Starting and Stopping JMS Bridges

- Starting and Stopping Links in a JMS Bridge

JMS Bridge Components

A JMS bridge consists of two primary components:

- One or more links, each of which maps between a destination in the Message Queue broker and a destination in an external JMS provider or in another Message Queue broker

- A default Dead Message Queue (DMQ) where undeliverable messages are sent. Additional, special-purpose DMQs can also be specified.

To provide destination mapping, each link consists of:

- A source: the destination from which the JMS bridge receives messages. The source consists of a connection factory for creating connections to a JMS provider and a destination in that provider.

- A target: the destination to which the JMS bridge forwards messages received from the source. The target consists of a connection factory for creating

connections to a JMS provider and a destination in that provider. Additionally, a target can optionally specify a message transformer that alters messages from the source before forwarding them to the target destination.

Links are unidirectional. Links that have an external JMS provider or another Message Queue broker as their source are called inbound links, and links that have the Message Queue broker as their source are called outbound links.

To configure these components, you specify several `imq.bridge.`*bridgeName* broker properties, and you create an XML configuration file that specifies the links, sources, targets, connection factories, destinations, and DMQs in the bridge. This XML configuration file must conform to the JMS bridge DTD.

JMS Bridge Features

To provide flexible, high-performing message transfer between mapped destinations, a JMS bridge offers these features:

- Pooled, Shared, and Dedicated Connections
- Transactional Message Transfer
- JMS Bridges in High Availability (HA) Broker Clusters
- Message Transformation During Message Delivery
- JMSReplyTo Header Processing
- Dead Message Queue (DMQ) Processing

Pooled, Shared, and Dedicated Connections

A JMS bridge uses the connection factories configured for link sources, link targets, and DMQs to create connections to the Message Queue broker and the external JMS provider. When making connections, the JMS bridge follows these rules to determine whether to use a pooled connection, a shared connection, or a dedicated connection:

- For link source connections, the JMS bridge always uses a dedicated connection.
- For link target and DMQ connections, the JMS bridge uses:
  - A pooled connection if the link target's or DMQ's `stay-connected` attribute is `false` and the connection factory has no JMS client identifier configured.
  - A dedicated connection if the link target's or DMQ's `stay-connected` attribute is `true` or if the link target's or DMQ's `clientid` attribute is set
  - A shared connection in all other cases

Transactional Message Transfer

Each JMS bridge includes a built-in XA transaction coordinator that enables its links to be configured as "transacted" such that message transfer from the source to the target is done in a XA distributed transaction.

For a link to be configured as transacted, both the link source and link target must specify connection factories of type `javax.jms.XAConnectionFactory`.

The built-in XA transaction coordinator logs transaction records using the same type of store as the Message Queue broker in which the JMS bridge resides. For file-based transaction logging, the transaction log for a JMS bridge is located at:

`IMQ_VARHOME/instances/brokerInstance/bridges/`*bridgeName*`/txlog.`*bridgeNane*

For JDBC-based transaction logging, the built-in XA transaction coordinator uses the same JDBC store as the Message Queue broker in which the JMS bridge resides.

Resource Manager Registration and The Built-In XA Transaction Coordinator When a JMS bridge is started, it registers all its Resource Managers (RMs) with the built-in XA transaction coordinator so that the coordinator can identify specific RMs during recovery.

For connection factories, the JMS bridge only registers the factory as an RM if the factory's `multi-rm` attribute value is `false`. A connection factory with a `multi-rm` attribute value of `true` should have each of its RMs identified by a connection factory whose `multi-rm` attribute value of `false` in order to participate in transactional message transfer. Additionally, the JMS bridge uses a connection factory's `ref-name` attribute value as part of its RM name when registering RMs with the built-in XA transaction coordinator, so this attribute should not be changed between restarts of the bridge if transaction recovery is desired between restarts.

JMS Bridges in High Availability (HA) Broker Clusters

Message Queue supports JMS bridges in HA broker clusters, but some special constraints apply due to the inherent nature of HA broker clusters:

- Each JMS bridge must have a name that is unique across all the JMS bridges in all the brokers in the cluster.

- Each JMS bridge must have the same bridge configuration across all the brokers in the cluster.

- The `imq.bridge.enabled` broker property must have the same value across all brokers in the cluster.

- Before broker startup, the `imq.bridge.activelist` broker property for each broker lists only those JMS bridges that are to be owned by that broker.

To ensure that bridges in the cluster have the same configuration across all brokers in the cluster, all bridge-related broker properties except for `imq.bridge.activelist` should be specified in the centralized cluster properties file defined by the `imq.cluster.url` broker property.

A table in the cluster's HA store is used to maintain a consistent record of JMS bridge ownership by the brokers in the cluster.

During broker startup, the JMS bridge service compares the broker's `imq.bridge.activelist` property value to this table's entries before starting any JMS bridges, with the following consequences:

- If a JMS bridge named in `imq.bridge.activelist` does not appear in the table, it is added to the table and associated with the broker.

- If a JMS bridge name in `imq.bridge.activelist` does appear in the table, and the table entry already associates the bridge with a different broker, the bridge name is removed from `imq.bridge.activelist`.

- If an entry in the table associates a JMS bridge with the broker, and that bridge's name is not in `imq.bridge.activelist`, the bridge name is added to `imq.bridge.activelist`.

Message Transformation During Message Delivery

A link target or a DMQ can specify a message transformer to process the message before it is delivered to the target destination or DMQ destination. This message transformer must be a Java class that extends the Message Queue bridge

`com.sun.messaging.bridge.service.MessageTransformer` abstract class and has `javax.jms.Message` as its formal type parameters. To specify a message transformer, set the `message-transformer-class` attribute of a link target or a DMQ to the fully qualified class name of the Java class.

During message transfer processing, the JMS message to be transferred to a target, plus any `property` subelements of the link's `target` element or the `dmq` element, are passed to the message transformer's `MessageTransformer.transform()` method, and the returned JMS message is sent to the target destination.

JMSReplyTo Header Processing

In a JMS message, the `JMSReplyTo` header value is provider dependent. Therefore, unless both the source provider and target provider are Message Queue, the JMS bridge sets an existing `JMSReplyTo` header to a null value before passing the message to a link target or, if specified, the message transformer for the link target.

This default behavior can be overridden by setting the `retain-replyto` attribute of the link's `target` element to `true`. Overriding the default behavior is useful when:

- The message transformer for a link target handles processing of the `JMSReplyTo` header.

- Both the link source and link target have the same JMS provider, and clients of the target provider instance need to send reply messages back across the JMS bridge to the `JMSReplyTo` destination in the source provider instance. To successfully implement this case:

  - The `JMSReplyTo` destination must exist (or be able to be auto-created) in the target provider instance.

  - A JMS bridge link must be defined with its source set to the `JMSReplyTo` destination in the target provider instance and its target set to the `JMSReplyTo` destination in the source provider instance.

Dead Message Queue (DMQ) Processing

Each JMS bridge includes a built-in Dead Message Queue (DMQ) named `built-in-dmq`. This DMQ is a designated Queue destination named `imq.bridge.jms.dmq` in the broker hosting the JMS bridge. You can also configure additional DMQs for the JMS bridge, in which case the DMQ can use any JMS destination in any configured JMS provider.

> **Note:** In a production environment, the built-in DMQ, `imq.bridge.jms.dmq`, should be administratively created and have its access controls set appropriately before starting a broker that uses JMS bridge services.

When a DMQ uses Message Queue as the JMS provider, it can be configured such that messages sent to it will automatically be transferred to the Message Queue broker's DMQ. To do so, set physical destination properties of the JMS bridge's DMQ as follows:

```
useDMQ=true
limitBehavior=REMOVE_OLDEST
maxNumMsgs=0
```

When a message is sent to the DMQ, the JMS bridge follows this sequence with the built-in DMQ first:

1. The bridge creates a new DMQ `javax.jms.ObjectMessage` object and sets the properties listed in Table 12–1 to the `ObjectMessage`.

2. If the DMQ has defined a message transformer, the original message is passed to the transformer's `MessageTransformer.transform()` method.

3. The body of the `javax.jms.ObjectMessage` is set to the transformed message (or original message if no message transformer is defined). If this action fails (usually because the message is not serializable), the body of the `ObjectMessage` is instead set to the `toString()` value of the original message.

4. The `javax.jms.ObjectMessage` is sent (up to `send-attempts` times) to the DMQ's destination with a `timeToLive` value based on the DMQ's `time-to-live-in-millis` attribute and with the same `JMSDeliveryMode` and `JMSPriority` as the original message.

5. If sending the message fails, the bridge repeats Steps 2 through 4 for each DMQ defined in the bridge's XML configuration file in the order they appear in the file, stopping when a send attempt succeeds, unless it is the built-in DMQ.

6. If the message can't be sent to any DMQ, a log message is generated, containing the properties and headers of the original message and the properties set in Step 1.

Table 12–1   DMQ Message Propeties

Property	Type	Description
JMS_SUN_JMSBRIDGE_DMQ_BODY_TRUNCATED	String	If unable to set the original message or the transformed message (if the DMQ has a message transformer) to the body of the DMQ `ObjectMessage`. In that case the message's `toString()` is set to the body of the DMQ `ObjectMessage`.
JMS_SUN_JMSBRIDGE_DMQ_EXCEPTION	String	The `Exception.getMessage()` if exception occurred or detailed comments on the failure; null if none.
JMS_SUN_JMSBRIDGE_DMQ_REASON	String	One of: `MESSAGE_EXPIRED`, `SEND_FAILURE`, `ACK_FAILURE`, `TRANSFORM_FAILURE`, `COMMIT_FAILURE`.
JMS_SUN_JMSBRIDGE_DMQ_TIMESTAMP	String	The timestamp when the JMS bridge sends the message to the DMQ.
JMS_SUN_JMSBRIDGE_SOURCE_CORRELATIONID	String	The original message's `getJMSCorrelationID()`.
JMS_SUN_JMSBRIDGE_SOURCE_DESTINATION	String	The original message's source destination name.
JMS_SUN_JMSBRIDGE_SOURCE_JMSTYPE	String	The original message's `getJMSType()`.
JMS_SUN_JMSBRIDGE_SOURCE_MESSAGEID	String	The orginal message's `getJMSMessageID()`, or null if not available.
JMS_SUN_JMSBRIDGE_SOURCE_PROVIDER	String	The `ConnectionMetaData.getJMSProviderName` of the connection the original message was received on; if not available, the source connection factory's `getClass().getName()`.
JMS_SUN_JMSBRIDGE_SOURCE_TIMESTAMP	Long	The original message's `getJMSTimestamp()`.
JMS_SUN_JMSBRIDGE_TARGET_DESTINATION	String	The name of the target destination where the original message was intended to send to.
JMS_SUN_JMSBRIDGE_TARGET_PROVIDER	String	The `ConnectionMetaData.getJMSProviderName` of the connection the original message was intended to send on; if not available, the target connection factory's `getClass().getName()`.

Message Processing Sequence Across a Link in a JMS Bridge

A JMS bridge link receives messages in the order sent by the link's source JMS provider and transfer them in the same order to the link's target JMS provider. A link follows this sequence to transfer each message:

1. The link receives a JMS message from its source.

2. The link checks the `JMSExpiration` header to determine whether the message has expired. If it has, a log message is generated, the JMS message is sent to the DMQ, and no further action is taken.

3. If the message has a `JMSReplyTo` header and the link target's `retain-replyto` attribute is `false`, the `JMSReplyTo` header value is set to null.

4. If the link target has defined a message transformer, the transformer's `MessageTransformer.transform()` method is called. If the call fails or if the message transformer returns a null value, a log message is generated and then processing continues as follows:

   - If the target's `consume-no-transfer-on-transform-error` XML attribute is `true`, the untransformed message is sent to the DMQ, consumed from the source, but not sent to the target.

   - If the target's `consume-no-transfer-on-transform-error` XML attribute is `false`, the link is stopped and the message is neither consumed from the source nor sent to the target.

5. If the `message-transfer-tag-bridge-name` attribute of the `jmsbridge` element is `true`, the `JMS_SUN_JMSBRIDGE_NAME` property is added to the message and set to the name of the bridge.

6. The message is sent to the link target's destination with a `timeToLive` value based on the `JMSExpiration` header and current GMT time and with the same `JMSDeliveryMode` and `JMSPriority` values as the original message. If sending to the link target's destination fails and the link is not transacted, a log message is generated, the JMS message is sent to the DMQ, and processing continues.

7. The source message is acknowledged using JMS `CLIENT_ACKNOWLEDGE` if the link is not transacted. If the acknowledgement fails, a log message is generated and the JMS message is sent to the DMQ.

8. If the message processing was successful, an INFO log message is generated. This log message can be suppressed by setting to `false` the `log-message-transfer` attribute of the `jmsbridge` element in the bridge's XML configuration file.

Message processing for messages across transacted links follows the same processing sequence, except JTA interfaces are used to coordinate the source and target resource managers to transfer the message in an XA distributed transaction. For transacted links, failure to send the message to the link target's destination does not cause the JMS message to be sent to the DMQ; instead, the transaction is rolled back. However, if the attempt to commit the transaction fails, a log message is generated and the JMS message is sent to the DMQ.

The quality of message transfer under failures depends on whether the link transferring the message is transacted:

- Transacted links: `once-only-once`

- Non-transacted links: `at-least-once`

Configuring a JMS Bridge

To configure a JMS bridge, you specify several `imq.bridge.`*bridgeName* broker properties, where *bridgeName* is a name you choose for the JMS bridge. Additionally, you create an XML configuration file that specifies the links, sources, targets, connection factories, destinations, and DMQs in the bridge. The url of this configuration file is provided as the value of the `imq.bridge.`*bridgeName*`.xml` property

Specifying the Broker Properties for a JMS Bridge

Table 12–2 lists the broker properties you can specify for a JMS bridge. In this table, *name* is the name of the JMS bridge, which must be unique across all bridges in the broker.

Table 12–2   Broker Properties for a JMS Bridge

Property	Type	Default Value	Description
`imq.bridge.`*name*`.type`	String	None	The bridge type of the bridge named *name*. For JMS bridges, specify a value of `JMS` or `jms`.
`imq.bridge.`*name*`.xmlurl`	String	None	The URL where the XML configuration file for the JMS bridge *name* is stored.
			Examples:
			■ `http://webserver/imq/jmsbridge1.config.xml`
			(for a file on a Web server)
			■ `file:/net/fileserver/imq/jmsbridge1.config.xml`
			(for a file on a shared drive)
`imq.bridge.`*name*`.autostart`	Boolean	`true`	Should the JMS bridge *name* be automatically started when the broker is started?
`imq.bridge.`*name*`.logfile.limit`	Integer	`0`	The approximate maximum number of bytes the JMS bridge *name* writes to any one log file.
			A value of `0` (zero) indicates that there is no maximum limit.
`imq.bridge.`*name*`.logfile.count`	Integer	`1`	The number of log files the JMS bridge *name* cycles through.
`imq.bridge.tm.props` `imq.bridge.`*name*`.tm.props`	String	None	Each of these properties specifies a list of key-value pairs for the built-in transaction coordinator for the JMS bridge *name*.
			The list consists of one or more *key=value* pairs separated by commas.
			When the `imq.persist.store` is `file`, the built-in transaction coordinator supports these keys: `txlogSize`, `txlogSync`, and `txlogMmap`.
			If the same key appears in both properties, the value specified in `imq.bridge.`*name*`.tm.props` takes precedence.

Creating the XML Configuration File for a JMS Bridge

As mentioned earlier, the XML configuration file for a JMS bridge specifies the following components of the bridge:

■   Links

■   Sources

- Targets

- Connection factories

- Destinations

- DMQs

The configuration file must conform to the JMS bridge DTD, which is stored at:

IMQ_HOME/lib/dtd/sun_jmsbridge_*Version*.dtd

The basic structure of configuration file is:

```
<jmsbridge name=bridgeName otherAttributes>
    <link name=linkName otherAttributes>
        [ <description>linkDescription</description> ]
        <source connection-factory-ref=connFactoryRef destination-ref=destRef
otherAttributes>
            [ <description>sourceDescription</description>
        </source>
        <target connection-factory-ref=connFactoryRef destination-ref=destRef
otherAttributes>
            [ <description>targetDescription</description> ]
            [ <property name=propName value=propValue /> ] ...
        </target>
    </link>
    <dmq name=dmqName otherAttributes> /* use name="built-in-dmq" for the built-in
DMQ */
        [ <description>dmqDescription</description> ]
        [ <property name=propName value=propValue /> ] ...
    </dmq>
    <connection-factory ref-name=connFactoryRef otherAttributes>
        [ <description>connFactoryDescription</description> ]
        [ <property name=propName value=propValue /> ] ...
    </connection-factory>
    <destination ref-name=destRef otherAttributes>
        [ <description>destDescription</description> ]
        [ <property name=propName value=propValue /> ] ...
    </destination> ...
</jmsbridge>
```

From this abbreviated structure for the bridge XML configuration file, note that source and target are subelements of link, while connection-factory and destination are peer elements to link, not subelements of source and target. Connection factories and destinations are associated with sources and targets by matching connection-factory ref-name and destination ref-name attributes values to source and target connection-factory-ref and destination-ref attribute values, respectively.

As a result of this association by name-matching instead of by subelement inclusion, you can use the same connection factories and destinations across sources and targets in multiple links, thus streamlining the configuration file and making it more manageable.

The following subsections describe the attributes you can specify for the elements in the JMS bridge XML configuration file.

jmsbridge Attributes Table 12–3 lists the attributes for the jmsbridge element in the JMS Bridge XML configuration file.

Table 12–3   jmsbridge Attributes

Attribute	Type	Description
name	String	Unique name for this JMS bridge.
		Default value: no default
log-message-transfer[1]	Boolean	Should each successful message transfer generate an INFO level log message?
		Default value: `true`
message-transfer-tag-bridge-name	Boolean	Should the `JMS_SUN_JMSBRIDGE_NAME` property be defined and set to the name of the bridge for each message before transferring to the link target?
		Default value: `false`

[1] First available in Message Queue 4.4.2

link Attributes Table 12–4 lists the attributes for the `link` element in the JMS Bridge XML configuration file.

Table 12–4   link Attributes

Attribute	Type	Description
enabled	Boolean	If `false`, the link will not be started
		Default value: `true`
name	String	Unique identifier for this link
		Default value: no default
transacted	Boolean	If `true`, each message transfer from source to target will be done in a XA distributed transaction. The connection factories specified by the source and target must be `javax.jms.XAConnectionFactory` objects.
		If `false`, CLIENT_ACKNOWLEDGE mode will be used on the source The connection factories specified by the source and target must be `javax.jms.ConnectionFactory` objects.
		Default value: `true`

source Attributes Table 12–5 lists the attributes for the `source` element in the JMS Bridge XML configuration file.

Table 12–5   source Attributes

Attribute	Type	Description
clientid	String	A JMS client identifier for the message consumer connection
		Default value: not set
connection-factory-ref	String	The `ref-name` attribute value of the `connection-factory` element to associate with this source.
		Default value: no default

Table 12–5 (Cont.) source Attributes

Attribute	Type	Description
destination-ref	String	The `ref-name` attribute value of the `destination` element to associate with this source. Default value: no default
durable-sub	String	A JMS durable subscription name. This attribute is ignored if the source's destination is not a `javax.jms.Topic` object. Default value: not set
selector	String	A JMS selector for the message consumer Default value: not set

`target` Attributes Table 12–6 lists the attributes for the `target` element in the JMS Bridge XML configuration file.

Table 12–6 target Attributes

Attribute	Type	Description
clientid	String	A JMS client identifier for the message producer connection; if set, use a dedicated connection. Default value: not set
connection-factory-ref	String	The `ref-name` attribute value of the `connection-factory` element to associate with this target. Default value: no default
consume-no-transfer-on-transform-error	Boolean	Controls processing when the message transformer's `MessageTransformer.transform()` method returns a null value or throws `java.lang.Throwable`: ■ If `true`, the message is sent to the DMQ and consumed from the source but not sent to the target. ■ If `false`, the link is stopped, and the message is neither consumed from the source nor transferred to the target. Default value: `false`
destination-ref	String	The `ref-name` attribute value of the `destination` element to associate with this target. The value `AS_SOURCE` is also supported. This value causes the target destination name and type to be set to the source message's `javax.jms.Message.getJMSDestination()`, unless overridden by the message transformer's `MessageTransformer.branchTo()`. Default value: no default

Table 12–6   (Cont.) target Attributes

Attribute	Type	Description
message-transformer-class	String	A fully qualified class name that extends the Message Queue bridge MessageTransformer class. For more information, see Message Transformation During Message Delivery. Place this class under the IMQ_HOME/lib/ext directory. Default value: not set
retain-replyto	Boolean	Should the value of the source message's JMSReplyTo header (if specified) be retained? For more information, see JMSReplyTo Header Processing. Default value: false
stay-connected	Boolean	If true, the message producer connection will stay connected, and be dedicated. Default value: true

dmq Attributes Table 12–7 lists the attributes for the dmq element in the JMS Bridge XML configuration file.

Table 12–7   dmq Attributes

Attribute	Type	Description
client-id	String	JMS client identifier for the DMQ producer connection. If set, the connection will be dedicated. Default value: not set
connection-factory-ref[1]	String	The ref-name attribute value of the connection-factory element to associate with this DMQ. This connection factory must be a javax.jms.ConnectionFactory object. Default value: no default
destination-ref[1]	String	The ref-name attribute value of the destination element to associate with this DMQ. Default value: no default
enabled[1]	Boolean	Is this DMQ is enabled? Default value: true
message-transformer-class	String	A fully qualified class name that extends the Message Queue bridge MessageTransformer class. For more information, see Message Transformation During Message Delivery. Default value: not set
name	String	The unique identifier of this DMQ. Default value: no default
send-attempt-interval-in-seconds	Integer	How long to wait before attempting to resend an undeliverable message to this DMQ. Default value: 5

Table 12–7   (Cont.) dmq Attributes

Attribute	Type	Description
send-attempts	Integer	The number of attempts to send (or resend) an undeliverable message to this DMQ. Default value: 3
stay-connected[1]	Boolean	If true, the DMQ producer connection will stay connected and be dedicated. Default value: true
time-to-live-in-millis	Integer	Time-to-live in milliseconds for messages going to this DMQ. The value 0 means forever. Default value: 0

[1] This attribute is ignored for the built-in DMQ, built-in-dmq

connection-factory Attributes Table 12–8 lists the attributes for the connection-factory element in the JMS Bridge XML configuration file.

Table 12–8   connection-factory Attributes

Attribute	Type	Description
connect-attempt-interval-in-seconds	Integer	How long to wait before each connect attempt. Default value: 5
connect-attempts	Integer	The number of attempts for connecting. The value -1 means retry forever Default value: -1
idle-timeout-in-seconds	Integer	Close a connection if it is idle for more than this long. The value 0 indicates no idle timeout. This attribute is ignored for sources and for targets and DMQs that have their stay-connected attribute set to true. Default value: 1800
lookup-name	String	JNDI lookup name. If specified, the JNDI environment properties must specified as property subelements of this connection-factory element. The object returned by the lookup must be either javax.jms.ConnectionFactory or javax.jms.XAConnectionFactory type If not specified, a default connection factory to the Message Queue broker hosting the bridge is created with the properties in the property subelements. Default value: not set
multi-rm	Boolean	Set to true if this connection factory will potentially create XA connections to more than one XA resource manager (that is, XAResource.isSame() is false among them). Also, add separate connection-factory for each such resource manager so that they will be registered separately to the built-in XA transaction coordinator. Default value: false

Table 12–8   (Cont.) connection-factory Attributes

Attribute	Type	Description
password	String	The password for the user specified in username. Default value: not set
ref-name	String	Unique name for this connection factory. Default value: no default
username	String	The user name to be used to create connections from this connection factory. If this attribute is set, the password attribute must also be set. If not set, connections are created using the no-argument createConnection() method of the connection factory. Default value: not set

destination Attributes Table 12–9 lists the attributes for the destination element in the JMS Bridge XML configuration file.

Table 12–9   destination Attributes

Attribute	Type	Description
lookup-name	String	JNDI lookup name for the destination. If specified, the JNDI environment properties must specified as property subelements of this destination element. Default value: not set
name	String	The JMS destination name of this destination. This attribute is ignored if lookup-name is specified. Default value: not set
ref-name	String	Unique name for this destination. Default value: no default
type	queue or topic	The JMS destination type of this destination. This attribute is ignored if lookup-name is specified Default value: queue

Starting and Stopping JMS Bridges

JMS bridges can be started automatically when the broker hosting the bridge starts, or manually using the imqbridgemgr utility. Similarly, JMS bridges are stopped automatically when the broker hosting the bridge is shut down, or manually using the imqbridgemgr utility.

When a JMS bridge is started, the JMS bridge manager performs these tasks:

1. Parse and validate the bridge's XML configuration file.

2. Initialize all links and DMQs that have their enabled attribute set to true.

3. If any enabled links have their transacted attribute set to true:

    a. Initialize the built-in XA transaction coordinator.

    b. Register resource managers (RMs) for all potential XA connection factories.

    c. Perform XA recovery for available RMs.

4. Create connection pools and shared connections as needed.

5. Ensure that all DMQs are ready.

6. Start all enabled links.

When a JMS bridge is stopped, the JMS bridge manager performs these tasks:

1. Stop all started and paused links.

2. Wait until all pooled connections are returned to their respective pools and until all references to shared connections are returned.

3. Close all connection pools and shared connections. This effectively causes all physical connections to JMS providers to close.

To Configure a JMS Bridge to Start at Broker Startup

Follow these steps to configure a JMS bridge so that it starts automatically when the broker hosting it is started.

1. Confirm that the bridge service manager is enabled.

See To Enable the Bridge Service Manager for instructions.

2. Add the name of the bridge to the `imq.bridge.activelist` broker property.

3. Confirm that the `imq.bridge.`*bridgeName*`.autostart` broker property is set to `true`.

To Start a JMS Bridge Manually

1. Enter the `imqbridgemgr start bridge` command, specifying the bridge name and the broker.

For example, to start the bridge `mq2external` hosted by the broker running on `myhost:8886`, enter this command:

```
imqbridgemgr start bridge -bn mq2external -b myhost:8886
```

To Stop a JMS Bridge Manually

1. Enter the `imqbridgemgr stop bridge` command, specifying the bridge name and the broker.

For example, to stop the bridge `mq2external` hosted by the broker running on `myhost:8886`, enter this command:

```
imqbridgemgr stop bridge -bn mq2external -b myhost:8886
```

Starting and Stopping Links in a JMS Bridge

Links in a JMS bridge are started automatically when the bridge starts. Similarly, links are stopped automatically when the bridge is stopped. Additionally, once a JMS bridge is started, any of its links can be paused, resumed, stopped or restarted manually using the `imqbridgemgr` utility.

To Stop a Link Manually

1. Enter the `imqbridgemgr stop link` command, specifying the link name, the bridge name, and the broker.

For example, to stop the link `link1` in the bridge `mq2external` hosted by the broker running on `myhost:8886`, enter this command:

```
imqbridgemgr stop link  -ln link1 -bn mq2external -b myhost:8886
```

To Start a Link Manually

A link cannot be started manually unless it is enabled; that is, the `enabled` attribute of its `link` element in the bridge's XML configuration file is set to `true`.

1. Enter the `imqbridgemgr start link` command, specifying the link name, the bridge name, and the broker.

   For example, to start the link `link1` in the bridge `mq2external` hosted by the broker running on `myhost:8886`, enter this command:

   ```
   imqbridgemgr start link  -ln link1 -bn mq2external -b myhost:8886
   ```

Configuring and Managing STOMP Bridge Services

The STOMP (Streaming Text Oriented Messaging Protocol) open source project at `http://docs.codehaus.org/display/STOMP/Home` defines a simple wire protocol that clients written in any language can use to communicate with any messaging provider that supports the STOMP protocol.

Message Queue 5.0 provides support for the STOMP protocol through the STOMP bridge service. This service enables a Message Queue broker to communicate with STOMP clients.

The STOMP bridge service provides the features needed to fully integrate STOMP messaging into the JMS messaging environment of Message Queue:

- Registration with the Message Queue Port Mapper service so that STOMP clients can discover the service dynamically

- Support for TCP and SSL/TLS connections, including SSL/TLS connections requiring client authentication

- Automatic conversion of STOMP frame messages to and from JMS `BytesMessage` and `TextMessage` types

- Extensible message handling and transformation (by defining a custom message transformer)

- Support for the full STOMP protocol, including the STOMP JMS bindings

The following subsections provide information about the STOMP bridge and how to configure and manage it:

- Configuring the STOMP Bridge

- Starting and Stopping the STOMP Bridge

- Message Processing Sequence Across the STOMP Bridge

- STOMP Protocol Features and the STOMP Bridge

Configuring the STOMP Bridge

To configure the STOMP bridge, you specify several `imq.bridge.stomp` broker properties in the broker hosting the bridge. These properties, which control the various features of the STOMP bridge, are listed in Table 12–10.

Table 12–10   Broker Properties for the STOMP Bridge Service

Property	Type	Default Value	Description
`imq.bridge.stomp.hostname`	String	None	Host name or IP address for the STOMP bridge service
			If specified, overrides `imq.hostname` (see Table 17–1) for the STOMP bridge service.
`imq.bridge.stomp.tcp.enabled`	Boolean	`true`	Does the STOMP bridge accept TCP connections?
`imq.bridge.stomp.tcp.port`	Integer	`7672`	The port on which the STOMP bridge listens for TCP connections, provided that `imq.bridge.stomp.tcp.enabled` is `true`.
`imq.bridge.stomp.tls.enabled`	Boolean	`false`	Does the STOMP bridge accept SSL/TLS connections?
			If `true`, a keystore must be created using the `imqkeytool` utility before starting the broker.
`imq.bridge.stomp.tls.port`	Integer	`7673`	The port on which the STOMP bridge listens for SSL/TLS connections, provided that `imq.bridge.stomp.tls.enabled` is `true`.
`imq.bridge.stomp.tls.requireClientAuth`	Boolean	`false`	Do SSL/TLS connections require client authentication?
`imq.bridge.stomp.consumerFlowLimit`	Integer	`1000`	The maximum number of unacknowledged messages that the STOMP bridge will deliver on a transacted STOMP subscription. The STOMP client must then acknowledge the messages and commit the transaction.
`imq.bridge.stomp.messageTransformer`	String	None	The fully qualified class name of a class that extends the Message Queue bridge `MessageTransformer` abstract class (with formal type parameters as `javax.jms.Message`). Place this class under the `IMQ_HOME/lib/ext` directory.
`imq.bridge.stomp.logfile.limit`	Integer	`0`	The approximate maximum number of bytes the STOMP bridge writes to any one log file.
			A value of `0` (zero) indicates that there is no maximum limit.
`imq.bridge.stomp.logfile.count`	Integer	`1`	The number of log files the STOMP bridge cycles through.

Starting and Stopping the STOMP Bridge

The STOMP bridge is started automatically when the broker hosting the bridge starts. Similarly, the STOMP bridge is stopped automatically when the broker hosting it is stopped. The STOMP bridge can be stopped and restarted manually using the `imqbridgemgr` utility.

To Activate the STOMP Bridge

1. Confirm that the bridge service manager is enabled.

   See To Enable the Bridge Service Manager for instructions.

2. Add the name `stomp` to the list of bridge names in the `imq.bridge.activelist` broker property.

To Stop the STOMP Bridge Manually

1. Enter the `imqbridgemgr stop bridge` command, specifying the bridge type and the broker.

   For example, to stop the STOMP bridge hosted by the broker running on `myhost:8886`, enter this command:

   ```
   imqbridgemgr stop bridge -t STOMP -b myhost:8886
   ```

To Start the STOMP Bridge Manually

1. Enter the `imqbridgemgr start bridge` command, specifying the bridge type and the broker.

   For example, to start the STOMP bridge hosted by the broker running on `myhost:8886`, enter this command:

   ```
   imqbridgemgr start bridge -t STOMP -b myhost:8886
   ```

Message Processing Sequence Across the STOMP Bridge

The STOMP bridge processes messages differently depending on whether the message is a STOMP frame message being received from a STOMP client or a JMS message being sent to a STOMP client.

For STOMP frame messages received from a STOMP client, the STOMP bridge performs these tasks:

1. Convert the STOMP frame message to a JMS `BytesMessage` if the `content-length` header is present; otherwise, convert it to a JMS `TextMessage` using UTF-8 as the message encoding.

2. If a custom message transformer is defined for the bridge, pass the JMS message to the transformer's `MessageTransformer.transform()` method.

3. Send the message to its destination.

For JMS messages sent to a STOMP client, the STOMP bridge performs these tasks:

1. If a custom message transformer is defined for the bridge, pass the JMS message to the transformer's `MessageTransformer.transform()` method.

2. If the transformed message (or original message when no custom transformer is defined) is not a JMS `TextMessage` or JMS `BytesMessage` message, close the STOMP connection and stop processing the message.

3. Convert the JMS message to a STOMP frame message, using UTF-8 encoding for all headers and for the message body of a JMS TextMessage message.

4. Send the message to the STOMP client.

Message Transformation During Message Processing

The message transformation between STOMP frame messages and JMS messages that the STOMP bridge automatically provides is sufficient in most applications. However, if you need to perform special processing or to send JMS message types other than

`BytesMessage` or `TextMessage` to STOMP clients, you can define a custom message transformer for the STOMP bridge.

This custom message transformer is a Java class that extends the Message Queue Bridge `MessageTransformer` abstract class by implementing the class's `transform()` method. Then, place the class file in the `IMQ_HOME/lib/ext` directory and set the `imq.bridge.stomp.messageTransformer` broker property of the broker hosting the STOMP bridge to the fully qualified class name of the class.

When implementing the `transform()` method, keep these points in mind:

- The formal parameters *T* and *S* must be of type `javax.jms.Message`.

- "The *source* and *target* arguments will be either `"STOMP"` and `"SUN_MQ"` or `"SUN_MQ"` and `"STOMP"`, respectively.

- A *source* argument value of `"STOMP"` indicates that the *message* argument is from a STOMP client SEND frame received by the STOMP bridge.

- A *source* argument value of `"SUN_MQ"` indicates that the *message* argument is from a Message Queue destination.

- The *readOnly* argument will be `false` if the *source* argument is `"STOMP"` and `true` if the *source* argument is `"SUN_MQ"`.

- If the *source* argument is `"STOMP"`, the *properties* argument contains, as key/value pairs, any arbitrary user headers that the STOMP bridge was unable to convert to JMS message properties in the *message* argument. Otherwise, the *properties* argument is null.

- The *charsetName* argument should be ignored unless the *source* argument is `"STOMP"` and the *message* argument is a JMS `BytesMessage` message. This combination of argument values indicates that the message is from a STOMP client and has already been converted to a `BytesMessage` message.

- The returned message must be in write-only mode if the *source* argument is `"STOMP"` and in read-only mode if the *source* argument is `"SUN_MQ"`.

STOMP Protocol Features and the STOMP Bridge

The STOMP bridge supports the full STOMP protocol, including all additional STOMP headers for the STOMP JMS bindings, as listed at
http://docs.codehaus.org/display/STOMP/Stomp+JMS.

The following table clarifies how the STOMP bridge handles certain command and header combinations that might be otherwise be subject to multiple interpretations.

Table 12–11   STOMP Bridge Handling of Selected Command/Header Combinations

STOMP Frame Command	STOMP Frame Header	Handling by the STOMP Bridge
CONNECT	login passcode	The STOMP bridge requires these headers to be specified; otherwise, it returns an ERROR frame.
SEND SUBSCRIBE UNSUBSCRIBE MESSAGE	destination	MQ STOMP bridge interprets prefixes in destination header values as follows: ■ /queue/: the prefix is followed by the name of a Queue ■ /topic/: the prefix is followed by the name of a Topic ■ /temp-queue/: the prefix is followed by the name of a TemporaryQueue ■ /temp-topic/: the prefix is followed by the name of a TemporaryTopic Note that the following two prefixes are reserved to be used only for send reply messages to a MESSAGE frame's replyto destination, and should only be used in the same CONNECT session in which the MESSAGE is received. ■ /temp-queue/temporary_destination://queue/ ■ /temp-topic/temporary_destination://topic/
SEND	expires priority persistent	When these headers are not specified for SEND, the message will be sent with the same default values as for a Message Queue Java client.
SEND	user specific headers	On SEND, a user can specify additional headers beyond the ones specified in the STOMP protocol and STOMP JMS Bindings. These headers are transformed to String properties of the converted JMS message. Therefore, the keys for these headers must be valid JMS property names. If any are not, and a custom message transformer is specified for STOMP bridge, the invalid ones are passed in the *properties* argument to the transformer's transform() method.
SUBSCRIBE	selector	Supported as described in the STOMP JMS Bindings on SUBSCRIBE.
SUBSCRIBE	id	A STOMP client should always specify an id header for SUBSCRIBE. If no "id" header is specified, the STOMP bridge assigns it a default value of /subscription-to/*STOMP-destination-name*. All SUBSCRIBE id values must be unique in the scope of the STOMP client connection to the STOMP bridge; otherwise, an ERROR frame will be returned.
SUBSCRIBE	transaction	For a STOMP subscription to receive messages in a transaction, the SUBSCRIBE frame must specify a transaction header with a transaction identifier whose transaction state is started. If the transaction does not exist, an ERROR frame is returned. After the transaction completes (using either COMMIT or ABORT), message delivery to the transacted subscription is paused until the next transaction BEGIN.
ABORT	transaction	For transacted subscriptions, aborting a transaction will cause the STOMP bridge to stop message delivery to all transacted subscriptions in the CONNECT session. Then, upon the next BEGIN, the STOMP bridge restarts the message delivery sequence to all the transacted subscriptions in the CONNECT session, including all unconsumed messages that had been previously delivered.

Table 12–11   (Cont.) STOMP Bridge Handling of Selected Command/Header Combinations

STOMP Frame Command	STOMP Frame Header	Handling by the STOMP Bridge
SUBSCRIBE	ack	For STOMP `ack:auto` (the default), a subscribed message is considered acknowledged as soon as it is sent to the STOMP client.
UNSUBSCRIBE	durable-subscriber-name	Unsubscribes a durable subscription, with these provisions: ■ `destination` and `id` headers, if specfied, are ignored. ■ An ERROR frame is returned if the connection (CONNECT) has no `client-id`. If an active subscriber with the durable name exists on the connection, it is first closed, and then the durable subscriber is unsubscribed.
BEGIN	transaction	Transactions are at STOMP CONNECT session level. Nested transactions are not supported. On attempts to start a nested transaction, an ERROR frame is returned. The transaction identifier will also be used for SUBSCRIBE frame to create a transacted subscription.
ACK	subscription	ACK should always specify a `subscription` header specifying the subscription id that the message to be acked was delivered to. If a subcriber id is not specified, the STOMP bridge default subscription id prefix is used to find the first matching subscription id with the prefix to ack the message. If the subscription for the specified subscription id was not created as transacted, and a `transaction` header is specified for the ACK, an ERROR frame is returned; ACK on a message ID, if found, will ACK all earlier messages delivered to the subscription in addition to the message with the given message ID.

Table 12–11   (Cont.) STOMP Bridge Handling of Selected Command/Header Combinations

STOMP Frame Command	STOMP Frame Header	Handling by the STOMP Bridge
ACK	transaction	For transacted subscription, an ACK for a message ID automatically ACKs all ealier messages sent to the transacted subscription in addition to the message with the given message ID. For transacted subscription, a message is considered consumed only when it is explicitly or implicitly ACKed in a transaction and there is a subsequent successful COMMIT on that transaction. If the transaction header is not specified but the subscription header is specified and the subscription is a transacted subscription, the message is ACKed in the current transaction if there is a current transaction. If there is no current transaction, an ERROR frame is returned.
MESSAGE ERROR	content-length	The STOMP bridge always sets the content-length header for MESSAGE and ERROR frames sent to STOMP clients.
SEND MESSAGE	reply-to	The STOMP bridge permits SEND from different STOMP CONNECT sessions as well as from the same CONNECT session to send reply messages to a STOMP reply-to header of temporary destination: ■ In the same CONNECT session, when SUBSCRIBE and SEND reply, use the same temporary destination string that is used in the SEND's reply-to header. ■ In a different CONNECT session, upon receiving a MESSAGE with a reply-to header of a temporary destination, use the same temporary destination string in the MESSAGE's reply-to header to SEND a reply to the reply-to temporary destination. This technique can also be used for sending the reply message when in the same CONNECT session.

13

Monitoring Broker Operations

This chapter describes the tools you can use to monitor a broker and how you can get metrics data. The chapter has the following sections:

- Monitoring Services
- Introduction to Monitoring Tools
- Configuring and Using Broker Logging
- Using the Command Utility to Display Metrics Interactively
- Using the JMX Administration API
- Using the Java ES Monitoring Console
- Using the Message-Based Monitoring API

Reference information on specific metrics is available in Metrics Information Reference

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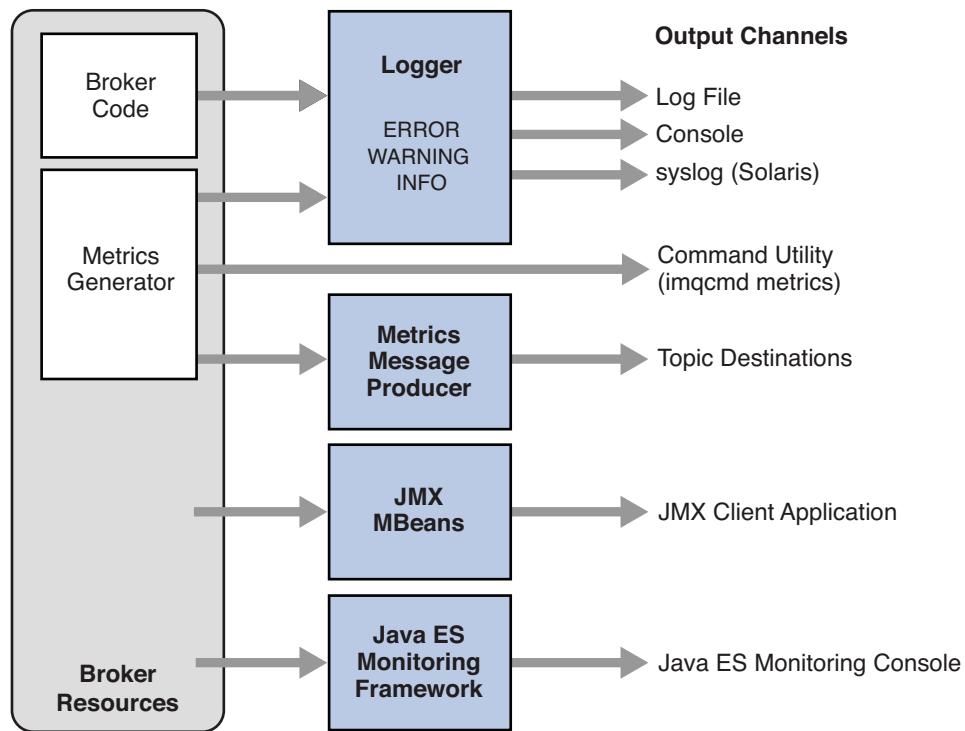
The broker includes components for monitoring and diagnosing application and broker performance. These include the components and services shown in the following figure:

- Broker code that logs broker events.
- A metrics generator that provides.

  The metrics generator provides information about broker activity, such as message flow in and out of the broker, the number of messages in broker memory and the memory they consume, the number of open connections, and the number of threads being used. The boolean broker property `imq.metrics.enabled` controls whether such information is logged and the `imq.metrics.interval` property specifies how often metrics information is generated.

- A logger component that writes out information to a number of output channels.
- A comprehensive set of Java Management Extensions (JMX) MBeans that expose broker resources using the JMX API
- Support for the Java ES Monitoring Framework
- A metrics message producer that sends JMS messages containing metrics information to topic destinations for consumption by JMS monitoring clients.

Broker properties for configuring the monitoring services are listed under Monitoring Properties.

Figure 13–1   Monitoring Services Support

Introduction to Monitoring Tools

There are five tools (or interfaces) for monitoring Message Queue information, as described briefly below:

- **Log files** provide a long-term record of metrics data, but cannot easily be parsed.

- **The Command Utility** (`imqcmd metrics`) lets you interactively sample information tailored to your needs, but does not provide historical information or allow you to manipulate the data programmatically.

- **The Java Management Extensions (JMX) Administration API** lets you perform broker resource configuration and monitoring operations programmatically from within a Java application. You can write your own JMX administration application or use the standard Java Monitoring and Management Console (`jconsole`).

- **The Sun Java Enterprise System Monitoring Framework (JESMF) and Monitoring Console** offers a common, Web-based graphical interface shared with other Java ES components, but can monitor only a subset of all Message Queue entities and operations.

- **The Message-based Monitoring API** lets you extract metrics information from messages produced by the broker to metrics topic destinations. However, to use it, you must write a Message Queue client application to capture, analyze, and display the metrics data.

The following tabel compares the different tools.

Table 13–1   Benefits and Limitations of Metrics Monitoring Tools

Metrics Monitoring Tool	Benefits	Limitations
Log files	- Regular sampling - Creates a historical record	- Local monitoring only - Data format difficult to read; no parsing tools - Need to configure broker properties; must shut down and restart broker to take effect - Broker metrics only; no destination or connection service metrics - No flexibility in selection of data - Same reporting interval for all metrics data; cannot be changed on the fly - Possible performance penalty if interval set too short
Command Utility (`imqcmd metrics`)	- Remote monitoring - Convenient for spot-checking - Data presented in easy-to-read tabular format - Easy to select specific data of interest - Reporting interval set in command option; can be changed on the fly	- Difficult to analyze data programmatically - No single command gets all data - No historical record; difficult to see historical trends
JMX Administration API	- Remote monitoring - Data can be analyzed programmatically and presented in any format - Easy to select specific data of interest - Can use standard Java Monitoring and Management Console (`jconsole`)	Might need to configure broker's JMX support
Java ES Monitoring Console	- Web-based graphical interface - Data presented in easy-to-read format - Common interface shared with other JES components - No performance penalty; pulls data from broker's existing data monitoring infrastructure	- Limited subset of data available - Data cannot be analyzed programmatically - No historical record; difficult to see historical trends
Message-based Monitoring API	- Remote monitoring - Data can be analyzed programmatically and presented in any format - Easy to select specific data of interest	- Need to configure broker properties; must shut down and restart broker to take effect - Same reporting interval for all metrics data; cannot be changed on the fly

In addition to the differences shown in the table, each tool gathers a somewhat different subset of the metrics information generated by the broker. For information on which metrics data is gathered by each monitoring tool, see Metrics Information Reference.

Configuring and Using Broker Logging

The Message Queue Logger takes information generated by broker code, a debugger, and a metrics generator and writes that information to a number of output channels: to standard output (the console), to a log file, and, on Solaris platforms, to the `syslog` daemon process. You can specify the type of information gathered by the Logger as well as the type of information the Logger writes to each of the output channels. For example, you can specify that you want metrics information written out to a log file.

This section describes the configuration and use of the Logger for monitoring broker activity. It includes the following topics:

- Logger Properties
- Log Message Format
- Default Logging Configuration
- Changing the Logging Configuration

Logger Properties

The `imq.log.file.dirpath` and `imq.log.file.filename` broker properties identify the log file to use and the `imq.log.console.stream` property specifies whether console output is directed to `stdout` or `stderr`.

The `imq.log.level` property controls the categories of metric information that the Logger gathers: `ERROR`, `WARNING`, or `INFO`. Each level includes those above it, so if you specify, for example, `WARNING` as the logging level, error messages will be logged as well.

There is also an `imq.destination.logDeadMsgs` property that specifies whether to log entries when dead messages are discarded or moved to the dead message queue.

The `imq.log.console.output` and `imq.log.file.output` properties control which of the specified categories the Logger writes to the console and the log file, respectively. In this case, however, the categories do *not* include those above them; so if you want, for instance, both errors and warnings written to the log file and informational messages to the console, you must explicitly set `imq.log.file.output` to `ERROR|WARNING` and `imq.log.console.output` to `INFO`.

On Solaris platforms another property, `imq.log.syslog.output`, specifies the categories of metric information to be written to the `syslog` daemon.

In the case of a log file, you can specify the point at which the file is closed and output is rolled over to a new file. Once the log file reaches a specified size (`imq.log.file.rolloverbytes`) or age (`imq.log.file.rolloversecs`), it is saved and a new log file created.

See Monitoring Properties for additional broker properties related to logging and subsequent sections for details about how to configure the Logger and how to use it to obtain performance information.

Log Message Format

A logged message consists of a time stamp, a message code, and the message itself. The volume of information included varies with the logging level you have set. The broker supports three logging levels: `ERROR`, `WARNING` , and `INFO` (see Table 13–2). Each level includes those above it (for example, `WARNING` includes `ERROR`).

Table 13–2   Logging Levels

Logging Level	Description
ERROR	Serious problems that could cause system failure
WARNING	Conditions that should be heeded but will not cause system failure
INFO	Metrics and other informational messages

The default logging level is INFO, so messages at all three levels are logged by default. 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
```

You can change the time zone used in the time stamp by setting the broker configuration property imq.log.timezone (see Table 17–13).

Default Logging Configuration

A broker is automatically configured to save log output to a set of rolling log files. The log files are located in a directory identified by the instance name of the associated broker:

IMQ_VARHOME/instances/*instanceName*/log

> **Note:**   For a broker whose life cycle is controlled by GlassFish Server, the log files are located in a subdirectory of the domain directory for the domain for which the broker was started:
>
> *domain-root-dir*/*domainName*/imq/instances/imqbroker/log

The log files are simple text files. The system maintains nine backup files named as follows, from earliest to latest:

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

By default, the log files are rolled over once a week. You can change this rollover interval, or the location or names of the log files, by setting appropriate configuration properties:

- 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.

- To change the frequency with which the log files are rolled over, set the property imq.log.file.rolloversecs.

See Table 17–13 for further information on these properties.

Changing the Logging Configuration

Log-related properties are described in Table 17–13.

To Change the Logger Configuration for a Broker

1. Set the logging 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. The following `imqbrokerd` options affect logging:

   -metrics *interval*
   Logging interval for broker metrics, in seconds

   -loglevel *level*
   Logging level (`ERROR`, `WARNING`, `INFO`, or `NONE`)

   -silent
   Silent mode (no logging to console)

   -tty
   Log all messages to console

   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

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 Logger output channels, you must make sure that logging is set at a level that supports the information you are mapping to the output channel. For example, if you set the logging 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 `WARNING` messages.

Changing Log File 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.

You can set or change the log file rollover properties when a broker is running. To set these properties, use the `imqcmd update bkr` command.

Sending Metrics Data to Log Files

This section describes the procedure for using broker log files to report metrics information. For general information on configuring the Logger, see Configuring and Using Broker Logging.

To Use Log Files to Report Metrics Information

1. Configure the broker's metrics generation capability:

   a. Confirm `imq.metrics.enabled=true`

      Generation of metrics for logging is turned on by default.

   b. Set the metrics generation interval to a convenient number of seconds.

      `imq.metrics.interval=`*interval*

      This value can be set in the `config.properties` file or using the `-metrics` *interval* command line option when starting up the broker.

2. Confirm that the Logger gathers metrics information:

   `imq.log.level=INFO`

This is the default value. This value can be set in the `config.properties` file or using the `-loglevel` *level* command line option when starting up the broker.

3. Confirm that the Logger is set to write metrics information to the log file:

```
imq.log.file.output=INFO
```

This is the default value. It can be set in the `config.properties` file.

4. Start up the broker.

The following shows sample broker metrics output to the log file:

```
[21/Jul/2004:11:21:18 PDT]
Connections: 0    JVM Heap: 8323072 bytes (7226576 free) Threads: 0 (14-1010)
       In: 0 msgs (0bytes) 0 pkts (0 bytes)
      Out: 0 msgs (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)
```

For reference information about metrics data, see Metrics Information Reference.

Logging Dead Messages

You can monitor physical destinations by enabling dead message logging for a broker. You can log dead messages whether or not you are using a dead message queue.

If you enable dead message logging, the broker logs the following types of events:

- A physical destination exceeded its maximum size.

- The broker removed a message from a physical destination, for a reason such as the following:

  - The destination size limit has been reached.

  - The message time to live expired.

  - The message is too large.

  - An error occurred when the broker attempted to process the message.

If a dead message queue is in use, logging also includes the following types of events:

- The broker moved a message to the dead message queue.

- The broker removed a message from the dead message queue and discarded it.

The following is an example of the log format for dead messages:

```
[29/Mar/2006:15:35:39 PST] [B1147]: Message 8-129.145.180.87(e7:6b:dd:5d:98:aa)-
35251-1143675279400 from destination Q:q0 has been placed on the DMQ because
[B0053]: Message on destination Q:q0 Expired: expiration time 1143675279402,
arrival time 1143675279401, JMSTimestamp 1143675279400
```

Dead message logging is disabled by default. To enable it, set the broker attribute `imq.destination.logDeadMsgs`.

Using the Command Utility to Display Metrics Interactively

A Message Queue broker can report metrics of the following types:

- **Java Virtual Machine (JVM) metrics**. Information about the JVM heap size.

- **Brokerwide metrics**. Information about messages stored in a broker, message flows into and out of a broker, and memory use. Messages are tracked in terms of numbers of messages and numbers of bytes.

- **Connection Service metrics**. Information about connections and connection thread resources, and information about message flows for a particular connection service.

- **Destination metrics**. Information about message flows into and out of a particular physical destination, information about a physical destination's consumers, and information about memory and disk space usage.

The `imqcmd` command can obtain metrics information for the broker as a whole, for individual connection services, and for individual physical destinations. To obtain metrics data, you generally use the `metrics` subcommand of `imqcmd`. Metrics data is written at an interval you specify, or the number of times you specify, to the console screen.

You can also use the `query` subcommand to view similar data that also includes configuration information. See imqcmd query for more information.

imqcmd metrics

The syntax and options of `imqcmd metrics` are shown in Table 13–3 and Table 13–4, respectively.

Table 13–3   `imqcmd metrics` Subcommand Syntax

Subcommand Syntax	Metrics Data Provided

`metrics bkr`
`[-b hostName:portNumber]`
`[-m metricType]`
`[-int interval]`
`[-msp numSamples]`
	Displays broker metrics for the default broker or a broker at the specified host and port.

`metrics svc -n serviceName`
`[-b hostName:portNumber]`
`[-m metricType]`
`[-int interval]`
`[-msp numSamples]`
	Displays metrics for the specified service on the default broker or on a broker at the specified host and port.

`metrics dst -t destType -n destName`
`[-b hostName:portNumber]`
`[-m metricType]`
`[-int interval]`
`[-msp numSamples]`
	Displays metrics information for the physical destination of the specified type and name.

Table 13–4   `imqcmd metrics` Subcommand Options

Subcommand Options	Description
`-b hostName:portNumber`	Specifies the hostname and port of the broker for which metrics data is reported. The default is `localhost:7676`. **Literal IP addresses as host names:** You can use a literal IPv4 or IPv6 address as a host name. If you use a literal IPv6 address, its format must conform to RFC2732 (http://www.ietf.org/rfc/rfc2732.txt), *Format for Literal IPv6 Addresses in URL's.*
`-int interval`	Specifies the interval (in seconds) at which to display the metrics. The default is 5 seconds.

Table 13–4   (Cont.) `imqcmd metrics` Subcommand Options

Subcommand Options	Description
-m *metricType*	Specifies the type of metric to display:
	ttl Displays metrics on messages and packets flowing into and out of the broker, service, or destination (default metric type).
	rts Displays metrics on rate of flow of messages and packets into and out of the broker, connection service, or destination (per second).
	cxn Displays connections, virtual memory heap, and threads (brokers and connection services only).
	con Displays consumer-related metrics (destinations only).
	dsk Displays disk usage metrics (destinations only).
-msp *numSamples*	Specifies the number of samples displayed in the output. The default is an unlimited number (infinite).
-n *destName*	Specifies the name of the physical destination (if any) for which metrics data is reported. There is no default.
-n *serviceName*	Specifies the connection service (if any) for which metrics data is reported. There is no default.
-t *destType*	Specifies the type (queue or topic) of the physical destination (if any) for which metrics data is reported. There is no default.

To Use the metrics Subcommand

1. Start the broker for which metrics information is desired.

   See Starting Brokers.

2. Issue the appropriate `imqcmd metrics` subcommand and options as shown in Table 13–3 and Table 13–4.

Metrics Outputs: imqcmd metrics

This section contains examples of output for the `imqcmd metrics` subcommand. The examples show brokerwide, connection service, and physical destination metrics.

Brokerwide Metrics

To get the rate of message and packet flow into and out of the broker at 10 second intervals, use the `metrics bkr` subcommand:

```
imqcmd metrics bkr -m rts -int 10 -u admin
```

This command produces output similar to the following (see data descriptions in Table 21–2):

```
-------------------------------------------------------
 Msgs/sec   Msg Bytes/sec  Pkts/sec   Pkt Bytes/sec
In    Out       In     Out     In    Out      In     Out
-------------------------------------------------------
0     0        27      56      0     0        38      66
10    0        7365    56      10    10       7457    1132
0     0        27      56      0     0        38      73
0     10       27      7402    10    20       1400    8459
0     0        27      56      0     0        38      73
```

Connection Service Metrics

To get cumulative totals for messages and packets handled by the jms connection service, use the `metrics svc` subcommand:

```
imqcmd metrics svc -n jms -m ttl -u admin
```

This command produces output similar to the following (see data descriptions in Table 21–3):

```
-------------------------------------------------
   Msgs       Msg Bytes       Pkts       Pkt Bytes
In   Out    In      Out    In   Out    In      Out
-------------------------------------------------
164  100  120704   73600   282  383  135967  102127
657  100  483552   73600   775  876  498815  149948
```

Physical Destination Metrics

To get metrics information about a physical destination, use the `metrics dst` subcommand:

```
imqcmd metrics dst -t q -n XQueue -m ttl -u admin
```

This command produces output similar to the following (see data descriptions in Table 21–4):

```
--------------------------------------------------------------------------------
   Msgs       Msg Bytes           Msg Count         Total Msg Bytes (k)    Largest
In   Out    In      Out     Current  Peak  Avg  Current  Peak     Avg     Msg (k)
--------------------------------------------------------------------------------
200  200  147200  147200      0      200    0     0      143      71        0
300  200  220800  147200    100      200   10    71      143      64        0
300  300  220800  220800      0      200    0     0      143      59        0
```

To get information about a physical destination's consumers, use the following `metrics dst` subcommand:

```
imqcmd metrics dst -t q -n SimpleQueue -m con -u admin
```

This command produces output similar to the following (see data descriptions in Table 21–4):

```
-------------------------------------------------------------------
   Active Consumers         Backup Consumers         Msg Count
Current  Peak  Avg      Current  Peak    Avg     Current  Peak  Avg
-------------------------------------------------------------------
   1      1     0          0      0       0        944    1000   525
```

imqcmd query

The syntax and options of `imqcmd query` are shown in Table 13–5 along with a description of the metrics data provided by the command.

Table 13–5   `imqcmd query` Subcommand Syntax

Subcommand Syntax	Metrics Data Provided
query bkr [-b *hostName*:*portNumber*]	Information on the current number of messages and message bytes stored in broker memory and persistent store (see Viewing Broker Information).

Table 13–5   (Cont.) `imqcmd` `query` Subcommand Syntax

Subcommand Syntax	Metrics Data Provided
`query svc -n serviceName` ` [-b hostName:portNumber]`	Information on the current number of allocated threads and number of connections for a specified connection service (see Viewing Connection Service Information).
`query dst -t destType -n destName` ` [-b hostName:portNumber]`	Information on the current number of producers, active and backup consumers, and messages and message bytes stored in memory and persistent store for a specified destination (see Viewing Physical Destination Information).

Note: Because of the limited metrics data provided by `imqcmd` `query` , this tool is not represented in the tables presented in Metrics Information Reference.

Using the JMX Administration API

The broker implements a comprehensive set of Java Management Extensions (JMX) MBeans that represent the broker's manageable resources. Using the JMX API, you can access these MBeans to perform broker configuration and monitoring operations programmatically from within a Java application.

In this way, the MBeans provide a Java application access to data values representing static or dynamic properties of a broker, connection, destination, or other resource. The application can also receive notifications of state changes or other significant events affecting the resource.

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

- You can include JMX code in your JMS client application to monitor application performance and, based on the results, to reconfigure the Message Queue resources you use to improve performance.

- You can write JMX client applications that monitor the broker to identify use patterns and performance problems, and you can use the JMX API to reconfigure the broker to optimize performance.

- You can write a JMX client application to automate regular maintenance tasks.

- You can write a JMX client application that constitutes your own version of the Command utility (`imqcmd`), and you can use it instead of `imqcmd`.

- You can use the standard Java Monitoring and Management Console (`jconsole`) that can provide access to the broker's MBeans.

For information on JMX infrastructure and configuring the broker's JMX support, see JMX Support. To manage a Message Queue broker using the JMX architecture, see the *Open Message Queue Developer's Guide for JMX Clients*.

Using the Java ES Monitoring Console

Message Queue supports the Sun Java System Monitoring Framework (JESMF), which allows Java Enterprise System (Java ES) components to be monitored using a common graphical interface, the Sun Java System Monitoring Console. Administrators can use the Monitoring Console to view performance statistics, create rules for automatic

monitoring, and acknowledge alarms. If you are running Message Queue along with other Java ES components, you may find it more convenient to use a single interface to manage all of them.

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

- The installed product

- The broker instance name

- The broker Port Mapper

- Each connection service

- Each physical destination

- The persistent data store

- The user repository

Each of these objects is mapped to a CMM object whose attributes can be monitored using the Java ES Monitoring Console. The reference tables in JES Monitoring Framework Reference identify those attributes that are available for JESMF monitoring. For detailed information about the mapping of Message Queue objects to CMM objects, see the *Sun Java Enterprise System Monitoring Guide*.

To enable JESMF monitoring, you must do the following:

1. Enable and configure the Monitoring Framework for all of your monitored components, as described in the *Sun Java Enterprise System Monitoring Guide*.

2. Install the Monitoring Console on a separate host, start the master agent, and then start the Web server, as described in the *Sun Java Enterprise System Monitoring Guide*.

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

For information on metric information provided by the Java ES Monitoring Framework, see JES Monitoring Framework Reference.

Using the Message-Based Monitoring API

Message Queue provides a Metrics Message Producer, which receives information from the Metrics Generator at regular intervals and writes the information into *metrics messages,*. The Metrics Message Producer then sends these messages to one of a number of metric topic destinations, depending on the type of metric information contained in the messages.

You can access this metrics information by writing a client application that subscribes to the metrics topic destinations, consumes the messages in these destinations, and processes the metrics information contained in the messages. This allows you to create custom monitoring tools to support messaging applications. For details of the metric quantities reported in each type of metrics message, see "Using the Metrics Monitoring API" in *Open Message Queue Developer's Guide for Java Clients*.

There are five metrics topic destinations, whose names are shown in Table 13–6, along with the type of metrics messages delivered to each destination.

Table 13–6   Metrics Topic Destinations

Topic Name	Description
`mq.metrics.broker`	Broker metrics
`mq.metrics.jvm`	Java Virtual Machine metrics
`mq.metrics.destination_list`	List of destinations and their types
`mq.metrics.destination.queue.`*queueName*	Destination metrics for queue *queueName*
`mq.metrics.destination.topic.`*topicName*	Destination metrics for topic *topicName*

The broker properties `imq.metrics.topic.enabled` and `imq.metrics.topic.interval` control, respectively, whether messages are sent to metric topic destinations and how often. The `imq.metrics.topic.timetolive` and `imq.metrics.topic.persist` properties specify the lifetime of such messages and whether they are persistent.

Besides the information contained in the body of a metrics message, the header of each message includes properties that provide the following additional information:

- The message type

- The address (host name and port number) of the broker that sent the message

- The time the metric sample was taken

These properties are useful to client applications that process metrics messages of different types or from different brokers.

Setting Up Message-Based Monitoring

This section describes the procedure for using the message-based monitoring capability to gather metrics information. The procedure includes both client development and administration tasks.

To Set Up Message-based Monitoring

1. Write a metrics monitoring client.

    See the *Open Message Queue Developer's Guide for Java Clients* for instructions on programming clients that subscribe to metrics topic destinations, consume metrics messages, and extract the metrics data from these messages.

2. Configure the broker's Metrics Message Producer by setting broker property values in the `config.properties` file:

    a. Enable metrics message production.

        Set `imq.metrics.topic.enabled=true`

        The default value is `true`.

    b. Set the interval (in seconds) at which metrics messages are generated.

        Set `imq.metrics.topic.interval=`*interval* .

        The default is 60 seconds.

    c. Specify whether you want metrics messages to be persistent (that is, whether they will survive a broker failure).

        Set `imq.metrics.topic.persist` .

        The default is `false`.

d. Specify how long you want metrics messages to remain in their respective destinations before being deleted.

Set `imq.metrics.topic.timetolive.`

The default value is 300 seconds.

3. Set any access control you desire on metrics topic destinations.

See the discussion in Security and Access Considerations below.

4. Start up your metrics monitoring client.

When consumers subscribe to a metrics topic, the metrics topic destination will automatically be created. Once a metrics topic has been created, the broker's metrics message producer will begin sending metrics messages to the metrics topic.

Security and Access Considerations

There are two reasons to restrict access to metrics topic destinations:

- Metrics data might include sensitive information about a broker and its resources.
- Excessive numbers of subscriptions to metrics topic destinations might increase broker overhead and negatively affect performance.

Because of these considerations, it is advisable to restrict access to metrics topic destinations.

Monitoring clients are subject to the same authentication and authorization control as any other client. Only users maintained in the Message Queue user repository are allowed to connect to the broker.

You can provide additional protections by restricting access to specific metrics topic destinations through an access control file, as described in User Authorization.

For example, the following entries in an `accesscontrol.properties` file will deny access to the mq.metrics.broker metrics topic to everyone except user1 and user 2.

```
topic.mq.metrics.broker.consume.deny.user=*
topic.mq.metrics.broker.consume.allow.user=user1,user2
```

The following entries will only allow users user3 to monitor topic t1.

```
topic.mq.metrics.destination.topic.t1.consume.deny.user=*
topic.mq.metrics.destination.topic.t1.consume.allow.user=user3
```

Depending on the sensitivity of metrics data, you can also connect your metrics monitoring client to a broker using an encrypted connection. For information on using encrypted connections, see Message Encryption.

Metrics Outputs: Metrics Messages

The metrics data outputs you get using the message-based monitoring API is a function of the metrics monitoring client you write. You are limited only by the data provided by the metrics generator in the broker. For a complete list of this data, see Metrics Information Reference.

14

Analyzing and Tuning a Message Service

This chapter covers a number of topics about how to analyze and tune a Message Queue service to optimize the performance of your messaging applications. It includes the following topics:

- About Performance
- Factors Affecting Performance
- Adjusting Configuration To Improve Performance

About Performance

This section provides some background information on performance tuning.

The Performance Tuning Process

The performance you get out of a messaging application depends on the interaction between the application and the Message Queue service. Hence, maximizing performance requires the combined efforts of both the application developer and the administrator.

The process of optimizing performance begins with application design and continues on through tuning the message service after the application has been deployed. The performance tuning process includes the following stages:

- Defining performance requirements for the application
- Designing the application taking into account factors that affect performance (especially tradeoffs between reliability and performance)
- Establishing baseline performance measures
- Tuning or reconfiguring the message service to optimize performance

The process outlined above is often iterative. During deployment of the application, a Message Queue administrator evaluates the suitability of the message service for the application's general performance requirements. If the benchmark testing meets these requirements, the administrator can tune the system as described in this chapter. However, if benchmark testing does not meet performance requirements, a redesign of the application might be necessary or the deployment architecture might need to be modified.

Aspects of Performance

In general, performance is a measure of the speed and efficiency with which a message service delivers messages from producer to consumer. However, there are several

different aspects of performance that might be important to you, depending on your needs.

Connection Load

The number of message producers, or message consumers, or the number of concurrent connections a system can support.

Message throughput

The number of messages or message bytes that can be pumped through a messaging system per second.

Latency

The time it takes a particular message to be delivered from message producer to message consumer.

Stability

The overall availability of the message service or how gracefully it degrades in cases of heavy load or failure.

Efficiency

The efficiency of message delivery; a measure of message throughput in relation to the computing resources employed.

These different aspects of performance are generally interrelated. If message throughput is high, that means messages are less likely to be backlogged in the broker, and as a result, latency should be low (a single message can be delivered very quickly). However, latency can depend on many factors: the speed of communication links, broker processing speed, and client processing speed, to name a few.

In any case, the aspects of performance that are most important to you generally depends on the requirements of a particular application.

Benchmarks

Benchmarking is the process of creating a test suite for your messaging application and of measuring message throughput or other aspects of performance for this test suite.

For example, you could create a test suite by which some number of producing clients, using some number of connections, sessions, and message producers, send persistent or nonpersistent messages of a standard size to some number of queues or topics (all depending on your messaging application design) at some specified rate. Similarly, the test suite includes some number of consuming clients, using some number of connections, sessions, and message consumers (of a particular type) that consume the messages in the test suite's physical destinations using a particular acknowledgment mode.

Using your standard test suite you can measure the time it takes between production and consumption of messages or the average message throughput rate, and you can monitor the system to observe connection thread usage, message storage data, message flow data, and other relevant metrics. You can then ramp up the rate of message production, or the number of message producers, or other variables, until performance is negatively affected. The maximum throughput you can achieve is a benchmark for your message service configuration.

Using this benchmark, you can modify some of the characteristics of your test suite. By carefully controlling all the factors that might have an effect on performance (see Application Design Factors Affecting Performance), you can note how changing some of these factors affects the benchmark. For example, you can increase the number of

connections or the size of messages five-fold or ten-fold, and note the effect on performance.

Conversely, you can keep application-based factors constant and change your broker configuration in some controlled way (for example, change connection properties, thread pool properties, JVM memory limits, limit behaviors, file-based versus JDBC-based persistence, and so forth) and note how these changes affect performance.

This benchmarking of your application provides information that can be valuable when you want to increase the performance of a deployed application by tuning your message service. A benchmark allows the effect of a change or a set of changes to be more accurately predicted.

As a general rule, benchmarks should be run in a controlled test environment and for a long enough period of time for your message service to stabilize. (Performance is negatively affected at startup by the just-in-time compilation that turns Java code into machine code.)

Baseline Use Patterns

Once a messaging application is deployed and running, it is important to establish baseline use patterns. You want to know when peak demand occurs and you want to be able to quantify that demand. For example, demand normally fluctuates by number of end users, activity levels, time of day, or all of these.

To establish baseline use patterns you need to monitor your message service over an extended period of time, looking at data such as the following:

- Number of connections

- Number of messages stored in the broker (or in particular physical destinations)

- Message flows into and out of a broker (or particular physical destinations)

- Numbers of active consumers

You can also use average and peak values provided in metrics data.

It is important to check these baseline metrics against design expectations. By doing so, you are checking that client code is behaving properly: for example, that connections are not being left open or that consumed messages are not being left unacknowledged. These coding errors consume broker resources and could significantly affect performance.

The base-line use patterns help you determine how to tune your system for optimal performance. For example:
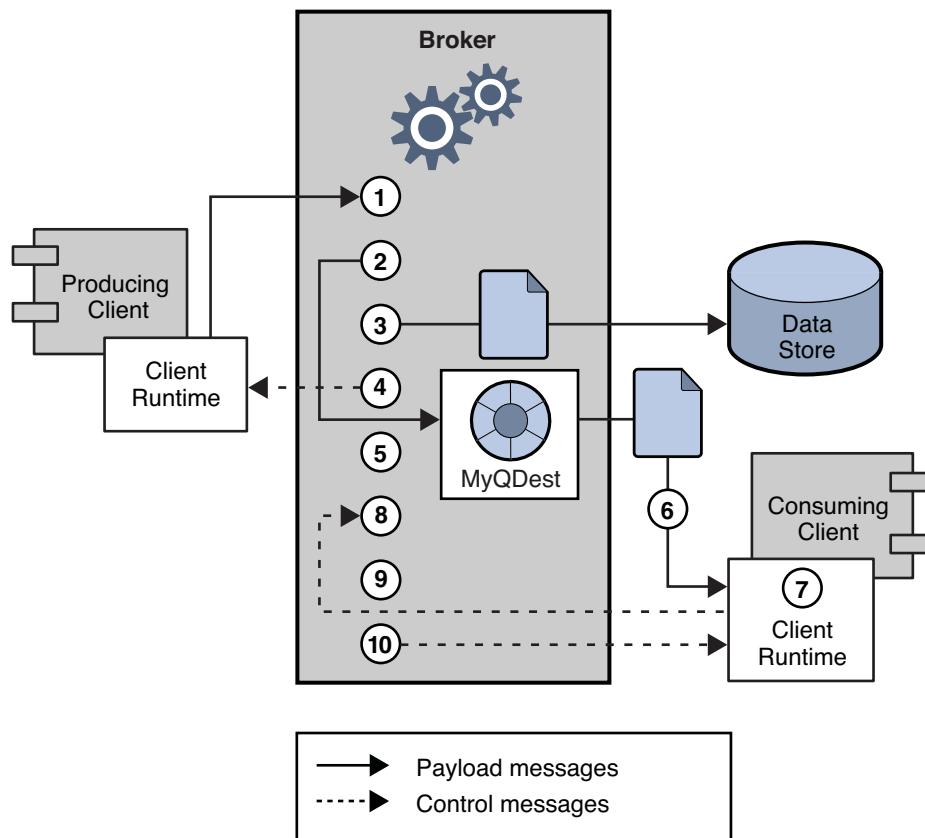
- If one physical destination is used significantly more than others, you might want to set higher message memory limits on that physical destination than on others, or to adjust limit behaviors accordingly.

- If the number of connections needed is significantly greater than allowed by the maximum thread pool size, you might want to increase the thread pool size or adopt a shared thread model.

- If peak message flows are substantially greater than average flows, that might influence the limit behaviors you employ when memory runs low.

In general, the more you know about use patterns, the better you are able to tune your system to those patterns and to plan for future needs.

Factors Affecting Performance

Message latency and message throughput, two of the main performance indicators, generally depend on the time it takes a typical message to complete various steps in the message delivery process. These steps are shown below for the case of a persistent, reliably delivered message. The steps are described following the illustration.

Figure 14–1   Message Delivery Through a Message Queue Service

Message Delivery Steps

1. The message is delivered from producing client to broker.

2. The broker reads in the message.

3. The message is placed in persistent storage (for reliability).

4. The broker confirms receipt of the message (for reliability).

5. The broker determines the routing for the message.

6. The broker writes out the message.

7. The message is delivered from broker to consuming client.

8. The consuming client acknowledges receipt of the message (for reliability).

9. The broker processes client acknowledgment (for reliability).

10. The broker confirms that client acknowledgment has been processed.

Since these steps are sequential, any one of them can be a potential bottleneck in the delivery of messages from producing clients to consuming clients. Most of the steps

depend on physical characteristics of the messaging system: network bandwidth, computer processing speeds, message service architecture, and so forth. Some, however, also depend on characteristics of the messaging application and the level of reliability it requires.

The following subsections discuss the effect of both application design factors and messaging system factors on performance. While application design and messaging system factors closely interact in the delivery of messages, each category is considered separately.

Application Design Factors Affecting Performance

Application design decisions can have a significant effect on overall messaging performance.

The most important factors affecting performance are those that affect the reliability of message delivery. Among these are the following:

- Delivery Mode (Persistent/Nonpersistent Messages)

- Use of Transactions

- Acknowledgment Mode

- Durable and Nondurable Subscriptions

Other application design factors affecting performance are the following:

- Use of Selectors (Message Filtering)

- Message Size

- Message Body Type

The sections that follow describe the effect of each of these factors on messaging performance. As a general rule, there is a tradeoff between performance and reliability: factors that increase reliability tend to decrease performance.

Table 14–1 shows how the various application design factors generally affect messaging performance. The table shows two scenarios—one high-reliability, low-performance, and one high-performance, low-reliability—and the choices of application design factors that characterize each. Between these extremes, there are many choices and tradeoffs that affect both reliability and performance.

Table 14–1   Comparison of High-Reliability and High-Performance Scenarios

Application Design Factor	High-Reliability, Low-Performance Scenario	High-Performance, Low-Reliability Scenario
Delivery mode	Persistent messages	Nonpersistent messages
Use of transactions	Transacted sessions	No transactions
Acknowledgment mode	`AUTO_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE`	`DUPS_OK_ACKNOWLEDGE`
Durable/nondurable subscriptions	Durable subscriptions	Nondurable subscriptions
Use of selectors	Message filtering	No message filtering
Message size	Large number of small messages	Small number of large messages
Message body type	Complex body types	Simple body types

Delivery Mode (Persistent/Nonpersistent Messages)

Persistent messages guarantee message delivery in case of broker failure. The broker stores the message in a persistent store until all intended consumers acknowledge they have consumed the message.

Broker processing of persistent messages is slower than for nonpersistent messages for the following reasons:

- A broker must reliably store a persistent message so that it will not be lost should the broker fail.

- The broker must confirm receipt of each persistent message it receives. Delivery to the broker is guaranteed once the method producing the message returns without an exception.

- Depending on the client acknowledgment mode, the broker might need to confirm a consuming client's acknowledgment of a persistent message.

For both queues and topics with durable subscribers, performance was approximately 40% faster for nonpersistent messages. We obtained these results using 10k-sized messages and `AUTO_ACKNOWLEDGE` mode.

Use of Transactions

A transaction is a guarantee that all messages produced in a transacted session and all messages consumed in a transacted session will be either processed or not processed (rolled back) as a unit.

Message Queue supports both local and distributed transactions.

A message produced or acknowledged in a transacted session is slower than in a nontransacted session for the following reasons:

- Additional information must be stored with each produced message.

- In some situations, messages in a transaction are stored when normally they would not be (for example, a persistent message delivered to a topic destination with no subscriptions would normally be deleted, however, at the time the transaction is begun, information about subscriptions is not available).

- Information on the consumption and acknowledgment of messages within a transaction must be stored and processed when the transaction is committed.

> **Note:** To improve performance, Message Queue message brokers are configured by default to use a memory-mapped file to store transaction data. On file systems that do not support memory-mapped files, you can disable this behavior by setting the broker property `imq.persist.file.transaction.memorymappedfile.enabled` to `false`.

Acknowledgment Mode

One mechanism for ensuring the reliability of JMS message delivery is for a client to acknowledge consumption of messages delivered to it by the Message Queue broker.

If a session is closed without the client acknowledging the message or if the broker fails before the acknowledgment is processed, the broker redelivers that message, setting a `JMSRedelivered` flag.

For a nontransacted session, the client can choose one of three acknowledgment modes, each of which has its own performance characteristics:

- AUTO_ACKNOWLEDGE. The system automatically acknowledges a message once the consumer has processed it. This mode guarantees at most one redelivered message after a provider failure.

- CLIENT_ACKNOWLEDGE. The application controls the point at which messages are acknowledged. All messages processed in that session since the previous acknowledgment are acknowledged. If the broker fails while processing a set of acknowledgments, one or more messages in that group might be redelivered.

- DUPS_OK_ACKNOWLEDGE. This mode instructs the system to acknowledge messages in a lazy manner. Multiple messages can be redelivered after a provider failure.

(Using CLIENT_ACKNOWLEDGE mode is similar to using transactions, except there is no guarantee that all acknowledgments will be processed together if a provider fails during processing.)

Acknowledgment mode affects performance for the following reasons:

- Extra control messages between broker and client are required in AUTO_ ACKNOWLEDGE and CLIENT_ACKNOWLEDGE modes. The additional control messages add additional processing overhead and can interfere with JMS payload messages, causing processing delays.

- In AUTO_ACKNOWLEDGE and CLIENT_ACKNOWLEDGE modes, the client must wait until the broker confirms that it has processed the client's acknowledgment before the client can consume additional messages. (This broker confirmation guarantees that the broker will not inadvertently redeliver these messages.)

- The Message Queue persistent store must be updated with the acknowledgment information for all persistent messages received by consumers, thereby decreasing performance.

Durable and Nondurable Subscriptions

Subscribers to a topic destination fall into two categories, those with durable and nondurable subscriptions.

Durable subscriptions provide increased reliability but slower throughput, for the following reasons:

- The Message Queue message service must persistently store the list of messages assigned to each durable subscription so that should a broker fail, the list is available after recovery.

- Persistent messages for durable subscriptions are stored persistently, so that should a broker fail, the messages can still be delivered after recovery, when the corresponding consumer becomes active. By contrast, persistent messages for nondurable subscriptions are not stored persistently (should a broker fail, the corresponding consumer connection is lost and the message would never be delivered).

We compared performance for durable and nondurable subscribers in two cases: persistent and nonpersistent 10k-sized messages. Both cases use AUTO_ACKNOWLEDGE acknowledgment mode. We found an effect on performance only in the case of persistent messages which slowed durables by about 30%

Use of Selectors (Message Filtering)

Application developers often want to target sets of messages to particular consumers. They can do so either by targeting each set of messages to a unique physical destination or by using a single physical destination and registering one or more selectors for each consumer.

A selector is a string requesting that only messages with property values that match the string are delivered to a particular consumer. For example, the selector `NumberOfOrders>1` delivers only the messages with a `NumberOfOrders` property value of `2` or more.

Creating consumers with selectors lowers performance (as compared to using multiple physical destinations) because additional processing is required to handle each message. When a selector is used, it must be parsed so that it can be matched against future messages. Additionally, the message properties of each message must be retrieved and compared against the selector as each message is routed. However, using selectors provides more flexibility in a messaging application.

Message Size

Message size affects performance because more data must be passed from producing client to broker and from broker to consuming client, and because for persistent messages a larger message must be stored.

However, by batching smaller messages into a single message, the routing and processing of individual messages can be minimized, providing an overall performance gain. In this case, information about the state of individual messages is lost.

In our tests, which compared throughput in kilobytes per second for 1k, 10k, and 100k-sized messages to a queue destination and `AUTO_ACKNOWLEDGE` acknowledgment mode, we found that nonpersistent messaging was about 50% faster for 1k messages, about 20% faster for 10k messages, and about 5% faster for 100k messages. The size of the message affected performance significantly for both persistent and nonpersistent messages. 100k messages are about 10 times faster than 10k, and 10k are about 5 times faster than 1k.

Message Body Type

JMS supports five message body types, shown below roughly in the order of complexity:

- `BytesMessage` contains a set of bytes in a format determined by the application.

- `TextMessage` is a simple Java string.

- `StreamMessage` contains a stream of Java primitive values.

- `MapMessage` contains a set of name-value pairs.

- `ObjectMessage` contains a Java serialized object.

While, in general, the message type is dictated by the needs of an application, the more complicated types (`MapMessage` and `ObjectMessage`) carry a performance cost: the expense of serializing and deserializing the data. The performance cost depends on how simple or how complicated the data is.

Message Service Factors Affecting Performance

The performance of a messaging application is affected not only by application design, but also by the message service performing the routing and delivery of messages.

The following sections discuss various message service factors that can affect performance. Understanding the effect of these factors is key to sizing a message service and diagnosing and resolving performance bottlenecks that might arise in a deployed application.

The most important factors affecting performance in a Message Queue service are the following:

- Hardware
- Operating System
- Java Virtual Machine (JVM)
- Connections
- Broker Limits and Behaviors
- Message Service Architecture
- Data Store Performance
- Client Runtime Configuration

The sections below describe the effect of each of these factors on messaging performance.

Hardware

For both the Message Queue broker and client applications, CPU processing speed and available memory are primary determinants of message service performance. Many software limitations can be eliminated by increasing processing power, while adding memory can increase both processing speed and capacity. However, it is generally expensive to overcome bottlenecks simply by upgrading your hardware.

Operating System

Because of the efficiencies of different operating systems, performance can vary, even assuming the same hardware platform. For example, the thread model employed by the operating system can have an important effect on the number of concurrent connections a broker can support. In general, all hardware being equal, Solaris is generally faster than Linux, which is generally faster than Windows.

Java Virtual Machine (JVM)

The broker is a Java process that runs in and is supported by the host JVM. As a result, JVM processing is an important determinant of how fast and efficiently a broker can route and deliver messages.

In particular, the JVM's management of memory resources can be critical. Sufficient memory has to be allocated to the JVM to accommodate increasing memory loads. In addition, the JVM periodically reclaims unused memory, and this memory reclamation can delay message processing. The larger the JVM memory heap, the longer the potential delay that might be experienced during memory reclamation.

Connections

The number and speed of connections between client and broker can affect the number of messages that a message service can handle as well as the speed of message delivery.

Broker Connection Limits All access to the broker is by way of connections. Any limit on the number of concurrent connections can affect the number of producing or consuming clients that can concurrently use the broker.

The number of connections to a broker is generally limited by the number of threads available. Message Queue can be configured to support either a dedicated thread model or a shared thread model (see Thread Pool Management).

The dedicated thread model is very fast because each connection has dedicated threads, however the number of connections is limited by the number of threads available (one input thread and one output thread for each connection). The shared thread model places no limit on the number of connections, however there is significant overhead and throughput delays in sharing threads among a number of connections, especially when those connections are busy.

Transport Protocols Message Queue software allows clients to communicate with the broker using various low-level transport protocols. Message Queue supports the connection services (and corresponding protocols) described in Configuring Connection Services.

The choice of protocols is based on application requirements (encrypted, accessible through a firewall), but the choice affects overall performance.

Figure 14–2   Transport Protocol Speeds

Our tests compared throughput for TCP and SSL for two cases: a high-reliability scenario (1k persistent messages sent to topic destinations with durable subscriptions and using AUTO_ACKNOWLEDGE acknowledgment mode) and a high-performance scenario (1k nonpersistent messages sent to topic destinations without durable subscriptions and using DUPS_OK_ACKNOWLEDGE acknowledgment mode).

In general we found that protocol has less effect in the high-reliability case. This is probably because the persistence overhead required in the high-reliability case is a more important factor in limiting throughput than the protocol speed. Additionally:

- TCP provides the fastest method to communicate with the broker.

- SSL is 50 to 70 percent slower than TCP when it comes to sending and receiving messages (50 percent for persistent messages, closer to 70 percent for nonpersistent messages). Additionally, establishing the initial connection is slower with SSL (it might take several seconds) because the client and broker (or Web Server in the case of HTTPS) need to establish a private key to be used when encrypting the data for transmission. The performance drop is caused by the additional processing required to encrypt and decrypt each low-level TCP packet.

- HTTP is slower than either the TCP or SSL. It uses a servlet that runs on a Web server as a proxy between the client and the broker. Performance overhead is involved in encapsulating packets in HTTP requests and in the requirement that messages go through two hops--client to servlet, servlet to broker--to reach the broker.

- HTTPS is slower than HTTP because of the additional overhead required to encrypt the packet between client and servlet and between servlet and broker.

Message Service Architecture

A Message Queue message service can be implemented as a single broker or as a cluster consisting of multiple interconnected broker instances.

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, thread, and memory limits. One way to accommodate increasing loads is to add more broker instances to a Message Queue message service, distributing client connections and message routing and delivery across multiple brokers.

In general, this scaling works best if clients are evenly distributed across the cluster, especially message producing clients. Because of the overhead involved in delivering messages between the brokers in a cluster, clusters with limited numbers of connections or limited message delivery rates, might exhibit lower performance than a single broker.

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

For more information on clusters, see Configuring and Managing Broker Clusters

Broker Limits and Behaviors

The message throughput that a broker might be required to handle is a function of the use patterns of the messaging applications the broker supports. However, the broker is limited in resources: memory, CPU cycles, and so forth. As a result, it would be possible for a broker to become overwhelmed to the point where it becomes unresponsive or unstable.

The Message Queue message broker has mechanisms built in for managing memory resources and preventing the broker from running out of memory. These mechanisms include configurable limits on the number of messages or message bytes that can be held by a broker or its individual physical destinations, and a set of behaviors that can be instituted when physical destination limits are reached.

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

Data Store Performance

Message Queue supports both file-based and JDBC-based persistence modules. File-based persistence uses individual files to store persistent data. JDBC-based persistence uses a Java Database Connectivity (JDBC) interface and requires a JDBC-compliant data store. File-based persistence is generally faster than JDBC-based; however, some users prefer the redundancy and administrative control provided by a JDBC-compliant store.

In the case of file-based persistence, you can maximize reliability by specifying that persistence operations synchronize the in-memory state with the data store. This helps eliminate data loss due to system crashes, but at the expense of performance.

Client Runtime Configuration

The Message Queue client runtime provides client applications with an interface to the Message Queue message service. It supports all the operations needed for clients to

send messages to physical destinations and to receive messages from such destinations. The client runtime is configurable (by setting connection factory attribute values), allowing you to control aspects of its behavior, such as connection flow metering, consumer flow limits, and connection flow limits, that can improve performance and message throughput. See Client Runtime Message Flow Adjustments for more information on these features and the attributes used to configure them.

Adjusting Configuration To Improve Performance

The following sections explain how configuration adjustments can affect performance.

System Adjustments

The following sections describe adjustments you can make to the operating system, JVM, communication protocols, and persistent data store.

Solaris Tuning: CPU Utilization, Paging/Swapping/Disk I/O

See your system documentation for tuning your operating system.

Java Virtual Machine Adjustments

By default, the broker uses a JVM heap size of 192MB. This is often too small for significant message loads and should be increased.

When the broker gets close to exhausting the JVM heap space used by Java objects, it uses various techniques such as flow control and message swapping to free memory. Under extreme circumstances it even closes client connections in order to free the memory and reduce the message inflow. Hence it is desirable to set the maximum JVM heap space high enough to avoid such circumstances.

However, if the maximum Java heap space is set too high, in relation to system physical memory, the broker can continue to grow the Java heap space until the entire system runs out of memory. This can result in diminished performance, unpredictable broker crashes, and/or affect the behavior of other applications and services running on the system. In general, you need to allow enough physical memory for the operating system and other applications to run on the machine.

In general it is a good idea to evaluate the normal and peak system memory footprints, and configure the Java heap size so that it is large enough to provide good performance, but not so large as to risk system memory problems.

To change the minimum and maximum heap size for the broker, use the `-vmargs` command line option when starting the broker. For example:

```
/usr/bin/imqbrokerd -vmargs "-Xms256m -Xmx1024m"
```

This command will set the starting Java heap size to 256MB and the maximum Java heap size to 1GB.

- On Solaris or Linux, if starting the broker via `/etc/rc*` (that is, `/etc/init.d/imq`), specify broker command line arguments in the file `/etc/imq/imqbrokerd.conf` (Solaris) or `/etc/opt/sun/mq/imqbrokerd.conf` (Linux). See the comments in that file for more information.

- On Windows, if starting the broker as a Window's service, specify JVM arguments using the `-vmargs` option to the `imqsvcadmin install` command. See Service Administrator Utility.

In any case, verify settings by checking the broker's log file or using the `imqcmd metrics bkr -m cxn` command.

Tuning Transport Protocols

Once a protocol that meets application needs has been chosen, additional tuning (based on the selected protocol) might improve performance.

A protocol's performance can be modified using the following three broker properties:

- `imq.protocol.`*protocolType*`.nodelay`

- `imq.protocol.`*protocolType*`.inbufsz`

- `imq.protocol.`*protocolType*`.outbufsz`

For TCP and SSL protocols, these properties affect the speed of message delivery between client and broker. For HTTP and HTTPS protocols, these properties affect the speed of message delivery between the Message Queue tunnel servlet (running on a Web server) and the broker. For HTTP/ HTTPS protocols there are additional properties that can affect performance (see HTTP/HTTPS Tuning).

The protocol tuning properties are described in the following sections.

nodelay The `nodelay` property affects Nagle's algorithm (the value of the `TCP_NODELAY` socket-level option on TCP/IP) for the given protocol. Nagle's algorithm is used to improve TCP performance on systems using slow connections such as wide-area networks (WANs).

When the algorithm is used, TCP tries to prevent several small chunks of data from being sent to the remote system (by bundling the data in larger packets). If the data written to the socket does not fill the required buffer size, the protocol delays sending the packet until either the buffer is filled or a specific delay time has elapsed. Once the buffer is full or the timeout has occurred, the packet is sent.

For most messaging applications, performance is best if there is no delay in the sending of packets (Nagle's algorithm is not enabled). This is because most interactions between client and broker are request/response interactions: the client sends a packet of data to the broker and waits for a response. For example, typical interactions include:

- Creating a connection

- Creating a producer or consumer

- Sending a persistent message (the broker confirms receipt of the message)

- Sending a client acknowledgment in an `AUTO_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE` session (the broker confirms processing of the acknowledgment)

For these interactions, most packets are smaller than the buffer size. This means that if Nagle's algorithm is used, the broker delays several milliseconds before sending a response to the consumer.

However, Nagle's algorithm may improve performance in situations where connections are slow and broker responses are not required. This would be the case where a client sends a nonpersistent message or where a client acknowledgment is not confirmed by the broker (`DUPS_OK_ACKNOWLEDGE` session).

inbufsz/outbufsz The `inbufsz` property sets the size of the buffer on the input stream reading data coming in from a socket. Similarly, `outbufsz` sets the buffer size of the output stream used by the broker to write data to the socket.

In general, both parameters should be set to values that are slightly larger than the average packet being received or sent. A good rule of thumb is to set these property values to the size of the average packet plus 1 kilobyte (rounded to the nearest kilobyte). For example, if the broker is receiving packets with a body size of 1 kilobyte, the overall size of the packet (message body plus header plus properties) is about 1200 bytes; an `inbufsz` of 2 kilobytes (2048 bytes) gives reasonable performance. Increasing `inbufsz` or `outbufsz` greater than that size may improve performance slightly, but increases the memory needed for each connection.

HTTP/HTTPS Tuning In addition to the general properties discussed in the previous two sections, HTTP/HTTPS performance is limited by how fast a client can make HTTP requests to the Web server hosting the Message Queue tunnel servlet.

A Web server might need to be optimized to handle multiple requests on a single socket. With JDK version 1.4 and later, HTTP connections to a Web server are kept alive (the socket to the Web server remains open) to minimize resources used by the Web server when it processes multiple HTTP requests. If the performance of a client application using JDK version 1.4 is slower than the same application running with an earlier JDK release, you might need to tune the Web server keep-alive configuration parameters to improve performance.

In addition to such Web server tuning, you can also adjust how often a client polls the Web server. HTTP is a request-based protocol. This means that clients using an HTTP-based protocol periodically need to check the Web server to see if messages are waiting. The `imq.httpjms.http.pullPeriod` broker property (and the corresponding `imq.httpsjms.https.pullPeriod` property) specifies how often the Message Queue client runtime polls the Web server.

If the `pullPeriod` value is `-1` (the default value), the client runtime polls the server as soon as the previous request returns, maximizing the performance of the individual client. As a result, each client connection monopolizes a request thread in the Web server, possibly straining Web server resources.

If the `pullPeriod` value is a positive number, the client runtime periodically sends requests to the Web server to see if there is pending data. In this case, the client does not monopolize a request thread in the Web server. Hence, if large numbers of clients are using the Web server, you might conserve Web server resources by setting the `pullPeriod` to a positive value.

Tuning the File-based Persistent Store

For information on tuning the file-based persistent store, see Configuring a File-Based Data Store.

Broker Memory Management Adjustments

You can improve performance and increase broker stability under load by properly managing broker memory. Memory management can be configured on a destination-by-destination basis or on a system-wide level (for all destinations, collectively).

Using Physical Destination Limits

To configure physical destination limits, see the properties described in Physical Destination Properties.

Using System-Wide Limits

If message producers tend to overrun message consumers, messages can accumulate in the broker. The broker contains a mechanism for throttling back producers and swapping messages out of active memory under low memory conditions, but it is wise to set a hard limit on the total number of messages (and message bytes) that the broker can hold.

Control these limits by setting the `imq.system.max_count` and the `imq.system.max_size` broker properties.

For example:

```
imq.system.max_count=5000
```

The defined value above means that the broker will only hold up to 5000 undelivered and/or unacknowledged messages. If additional messages are sent, they are rejected by the broker. If a message is persistent then the clinet runtime will throw an exception when the producer tries to send the message. If the message is non-persistent, the broker silently drops the message.

When an exception is thrown in sending a message, the client should process the exception by pausing for a moment and retrying the send again. (Note that the exception will never be due to the broker's failure to receive a message; the exception is thrown by the client runtime before the message is sent to the broker.)

Client Runtime Message Flow Adjustments

This section discusses client runtimeflow control behaviors that affect performance. These behaviors are configured as attributes of connection factory administered objects. For information on setting connection factory attributes, see Managing Administered Objects.

Message Flow Metering

Messages sent and received by clients *(payload messages),* as well as Message Queue control messages, pass over the same client-broker connection. Delays in the delivery of control messages, such as broker acknowledgments, can result if control messages are held up by the delivery of payload messages. To prevent this type of congestion, Message Queue meters the flow of payload messages across a connection.

Payload messages are batched (as specified with the connection factory attribute `imqConnectionFlowCount`) so that only a set number are delivered. After the batch has been delivered, delivery of payload messages is suspended and only pending control messages are delivered. This cycle repeats, as additional batches of payload messages are delivered followed by pending control messages.

The value of `imqConnectionFlowCount` should be kept low if the client is doing operations that require many responses from the broker: for example, if the client is using `CLIENT_ACKNOWLEDGE` or `AUTO_ACKNOWLEDGE` mode, persistent messages, transactions, or queue browsers, or is adding or removing consumers. If, on the other hand, the client has only simple consumers on a connection using `DUPS_OK_ACKNOWLEDGE` mode, you can increase `imqConnectionFlowCount` without compromising performance.

Message Flow Limits

There is a limit to the number of payload messages that the Message Queue client runtime can handle before encountering local resource limitations, such as memory. When this limit is approached, performance suffers. Hence, Message Queue lets you

limit the number of messages per consumer (or messages per connection) that can be delivered over a connection and buffered in the client runtime, waiting to be consumed.

Consumer Flow Limits When the number of payload messages delivered to the client runtime exceeds the value of `imqConsumerFlowLimit` for any consumer, message delivery for that consumer stops. It is resumed only when the number of unconsumed messages for that consumer drops below the value set with `imqConsumerFlowThreshold`.

The following example illustrates the use of these limits: consider the default settings for topic consumers:

```
imqConsumerFlowLimit=1000
imqConsumerFlowThreshold=50
```

When the consumer is created, the broker delivers an initial batch of 1000 messages (providing they exist) to this consumer without pausing. After sending 1000 messages, the broker stops delivery until the client runtime asks for more messages. The client runtime holds these messages until the application processes them. The client runtime then allows the application to consume at least 50% (`imqConsumerFlowThreshold`) of the message buffer capacity (i.e. 500 messages) before asking the broker to send the next batch.

In the same situation, if the threshold were 10%, the client runtime would wait for the application to consume at least 900 messages before asking for the next batch.

The next batch size is calculated as follows:

```
imqConsumerFlowLimit - (current number of pending msgs in buffer)
```

So if `imqConsumerFlowThreshold` is 50%, the next batch size can fluctuate between 500 and 1000, depending on how fast the application can process the messages.

If the `imqConsumerFlowThreshold` is set too high (close to 100%), the broker will tend to send smaller batches, which can lower message throughput. If the value is set too low (close to 0%), the client may be able to finish processing the remaining buffered messages before the broker delivers the next set, again degrading message throughput. Generally speaking, unless you have specific performance or reliability concerns, you will not need to change the default value of `imqConsumerFlowThreshold` attribute.

The consumer-based flow controls (in particular, `imqConsumerFlowLimit`) are the best way to manage memory in the client runtime. Generally, depending on the client application, you know the number of consumers you need to support on any connection, the size of the messages, and the total amount of memory that is available to the client runtime.

Note: Setting the `imqConsumerFlowLimitPrefetch` property to `false` disables the prefetching and buffering specified by `imqConsumerFlowLimit` and `imqConsumerFlowThreshold`, in which case messages are delivered to consumers one at a time and a new message is not sent to a consumer until it consumes the message it has. This delivery constraint, which can degrade message throughput, is for use when business logic demands that each consumer have only one message at a time.

When the JMS resource adapter, jmsra, is used to consume messages in a GlassFish Server cluster, this behavior is defined using different properties, as described in About Shared Topic Subscriptions for Clustered Containers.

Connection Flow Limits In the case of some client applications, however, the number of consumers may be indeterminate, depending on choices made by end users. In those cases, you can still manage memory using connection-level flow limits.

Connection-level flow controls limit the total number of messages buffered for *all* consumers on a connection. If this number exceeds the value of `imqConnectionFlowLimit`, delivery of messages through the connection stops until that total drops below the connection limit. (The `imqConnectionFlowLimit` attribute is enabled only if you set `imqConnectionFlowLimitEnabled` to `true`.)

The number of messages queued up in a session is a function of the number of message consumers using the session and the message load for each consumer. If a client is exhibiting delays in producing or consuming messages, you can normally improve performance by redesigning the application to distribute message producers and consumers among a larger number of sessions or to distribute sessions among a larger number of connections.

Adjusting Multiple-Consumer Queue Delivery

The efficiency with which multiple queue consumers process messages in a queue destination depends on a number of factors. To achieve optimal message throughput there must be a sufficient number of consumers to keep up with the rate of message production for the queue, and the messages in the queue must be routed and then delivered to the active consumers in such a way as to maximize their rate of consumption.

The message delivery mechanism for multiple-consumer queues is that messages are delivered to consumers in batches as each consumer is ready to receive a new batch. The readiness of a consumer to receive a batch of messages depends upon configurable client runtime properties, such as `imqConsumerFlowLimit` and `imqConsumerFlowThreshold`, as described in Message Flow Limits. As new consumers are added to a queue, they are sent a batch of messages to consume, and receive subsequent batches as they become ready.

> **Note:** The message delivery mechanism for multiple-consumer queues described above can result in messages being consumed in an order different from the order in which they are produced.

If messages are accumulating in the queue, it is possible that there is an insufficient number of consumers to handle the message load. It is also possible that messages are being delivered to consumers in batch sizes that cause messages to be backing up on the consumers. For example, if the batch size (`consumerFlowLimit`) is too large, one consumer might receive all the messages in a queue while other consumers receive none. If consumers are very fast, this might not be a problem. However, if consumers are relatively slow, you want messages to be distributed to them evenly, and therefore you want the batch size to be small. Although smaller batch sizes require more overhead to deliver messages to consumers, for slow consumers there is generally a net performance gain in using small batch sizes. The value of `consumerFlowLimit` can be set on a destination as well as on the client runtime: the smaller value overrides the larger one.

15

Troubleshooting

This chapter explains how to understand and resolve the following problems:

- A Client Cannot Establish a Connection

- Connection Throughput Is Too Slow

- A Client Cannot Create a Message Producer

- Message Production Is Delayed or Slowed

- Messages Are Backlogged

- Broker Throughput Is Sporadic

- Messages Are Not Reaching Consumers

- Dead Message Queue Contains Messages

When problems occur, it is useful to check the version number of the installed Message Queue software. Use the version number to ensure that you are using documentation whose version matches the software version. You also need the version number to report a problem to Oracle. To check the version number, issue the following command:

```
imqcmd -v
```

A Client Cannot Establish a Connection

Symptoms:

- Client cannot make a new connection.

- Client cannot auto-reconnect on failed connection.

Possible causes:

- Client applications are not closing connections, causing the number of connections to exceed resource limitations.

- Broker is not running or there is a network connectivity problem.

- Connection service is inactive or paused.

- Too few threads available for the number of connections required.

- Too few file descriptors for the number of connections required on the Solaris or Linux platform.

- TCP backlog limits the number of simultaneous new connection requests that can be established.

- [Operating system limits the number of concurrent connections..](#)

- [Authentication or authorization of the user is failing.](#)

Client applications are not closing connections, causing the number of connections to exceed resource limitations.

To confirm this cause of the problem: List all connections to a broker:

```
imqcmd list cxn
```

The output will list all connections and the host from which each connection has been made, revealing an unusual number of open connections for specific clients.

To resolve the problem: Rewrite the offending clients to close unused connections.

Broker is not running or there is a network connectivity problem.

To confirm this cause of the problem:

- Telnet to the broker's primary port (for example, the default of 7676) and verify that the broker responds with Port Mapper output.

- Verify that the broker process is running on the host.

To resolve the problem:

- Start up the broker.

- Fix the network connectivity problem.

Connection service is inactive or paused.

To confirm this cause of the problem: Check the status of all connection services:

```
imqcmd list svc
```

If the status of a connection service is shown as unknown or paused, clients will not be able to establish a connection using that service.

To resolve the problem:

- If the status of a connection service is shown as unknown , it is missing from the active service list (imq.service.active). In the case of SSL-based services, the service might also be improperly configured, causing the broker to make the following entry in the broker log:

  ```
  ERROR [B3009]: Unable to start service ssljms:[B4001]: Unable to open
  protocol tls for ssljms service...
  ```

  followed by an explanation of the underlying cause of the exception.

  To properly configure SSL services, see [Message Encryption](#).

- If the status of a connection service is shown as paused, resume the service (see [Pausing and Resuming a Connection Service](#)).

Too few threads available for the number of connections required.

To confirm this cause of the problem: Check for the following entry in the broker log:

```
WARNING [B3004]: No threads are available to process a new connection on
service ...Closing the new connection.
```

Also check the number of connections on the connection service and the number of threads currently in use, using one of the following formats:

```
imqcmd query svc -n serviceNameimqcmd metrics svc -n serviceName-m cxn
```

Each connection requires two threads: one for incoming messages and one for outgoing messages (see Thread Pool Management).

To resolve the problem:

- If you are using a dedicated thread pool model (imq.*serviceName*.threadpool_ model=dedicated), the maximum number of connections is half the maximum number of threads in the thread pool. Therefore, to increase the number of connections, increase the size of the thread pool (imq.*serviceName*.max_threads) or switch to the shared thread pool model.

- If you are using a shared thread pool model (imq.*serviceName*.threadpool_ model=shared), the maximum number of connections is half the product of the connection monitor limit (imq.*serviceName*.connectionMonitor_limit) and the maximum number of threads (imq.*serviceName*.max_threads). Therefore, to increase the number of connections, increase the size of the thread pool or increase the connection monitor limit.

- Ultimately, the number of supportable connections (or the throughput on connections) will reach input/output limits. In such cases, use a multiple-broker cluster to distribute connections among the broker instances within the cluster.

Too few file descriptors for the number of connections required on the Solaris or Linux platform.

For more information about this issue, see Setting the File Descriptor Limit.

To confirm this cause of the problem: Check for an entry in the broker log similar to the following:

```
Too many open files
```

To resolve the problem: Increase the file descriptor limit, as described in the man page for the ulimit command.

TCP backlog limits the number of simultaneous new connection requests that can be established.

The TCP backlog places a limit on the number of simultaneous connection requests that can be stored in the system backlog (imq.portmapper.backlog) before the Port Mapper rejects additional requests. (On the Windows platform there is a hard-coded backlog limit of 5 for Windows desktops and 200 for Windows servers.)

The rejection of requests because of backlog limits is usually a transient phenomenon, due to an unusually high number of simultaneous connection requests.

To confirm this cause of the problem: Examine the broker log. First, check to see whether the broker is accepting some connections during the same time period that it is rejecting others. Next, check for messages that explain rejected connections. If you find such messages, the TCP backlog is probably not the problem, because the broker does not log connection rejections due to the TCP backlog. If some successful connections are logged, and no connection rejections are logged, the TCP backlog is probably the problem.

To resolve the problem:

- Program the client to retry the attempted connection after a short interval of time (this normally works because of the transient nature of this problem).

- Increase the value of imq.portmapper.backlog.

- Check that clients are not closing and then opening connections too often.

Operating system limits the number of concurrent connections.

The Windows operating system license places limits on the number of concurrent remote connections that are supported.

To confirm this cause of the problem: Check that there are plenty of threads available for connections (using `imqcmd query svc`) and check the terms of your Windows license agreement. If you can make connections from a local client, but not from a remote client, operating system limitations might be the cause of the problem.

To resolve the problem:

- Upgrade the Windows license to allow more connections.

- Distribute connections among a number of broker instances by setting up a multiple-broker cluster.

Authentication or authorization of the user is failing.

The authentication may be failing for any of the following reasons:

- Incorrect password

- No entry for user in user repository

- User does not have access permission for connection service

To confirm this cause of the problem: Check entries in the broker log for the `Forbidden` error message. This will indicate an authentication error, but will not indicate the reason for it.

- If you are using a file-based user repository, enter the following command:

  `imqusermgr list -i` *instanceName*`-u` *userName*

  If the output shows a user, the wrong password was probably submitted. If the output shows the following error, there is no entry for the user in the user repository:

  `Error [B3048]: User does not exist in the password file`

- If you are using an LDAP server user repository, use the appropriate tools to check whether there is an entry for the user.

- Check the access control file to see whether there are restrictions on access to the connection service.

To resolve the problem:

- If the wrong password was used, provide the correct password.

- If there is no entry for the user in the user repository, add one (see Adding a User to the Repository).

- If the user does not have access permission for the connection service, edit the access control file to grant such permission (see Authorization Rules for Connection Services).

Connection Throughput Is Too Slow

Symptoms:

- Message throughput does not meet expectations.

- Message input/output rates are not limited by an insufficient number of supported connections (as described in A Client Cannot Establish a Connection).

Possible causes:

- Network connection or WAN is too slow.
- Connection service protocol is inherently slow compared to TCP.
- Connection service protocol is not optimally tuned.
- Messages are so large that they consume too much bandwidth.
- What appears to be slow connection throughput is actually a bottleneck in some other step of the message delivery process.

Network connection or WAN is too slow.

To confirm this cause of the problem:

- Ping the network, to see how long it takes for the ping to return, and consult a network administrator.
- Send and receive messages using local clients and compare the delivery time with that of remote clients (which use a network link).

To resolve the problem: Upgrade the network link.

Connection service protocol is inherently slow compared to TCP.

For example, SSL-based or HTTP-based protocols are slower than TCP (see Transport Protocols).

To confirm this cause of the problem: If you are using SSL-based or HTTP-based protocols, try using TCP and compare the delivery times.

To resolve the problem: Application requirements usually dictate the protocols being used, so there is little you can do other than attempt to tune the protocol as described in Tuning Transport Protocols.

Connection service protocol is not optimally tuned.

To confirm this cause of the problem: Try tuning the protocol to see whether it makes a difference.

To resolve the problem: Try tuning the protocol, as described in Tuning Transport Protocols.

Messages are so large that they consume too much bandwidth.

To confirm this cause of the problem: Try running your benchmark with smaller-sized messages.

To resolve the problem:

- Have application developers modify the application to use the message compression feature, which is described in the *Open Message Queue Developer's Guide for Java Clients*.
- Use messages as notifications of data to be sent, but move the data using another protocol.

What appears to be slow connection throughput is actually a bottleneck in some other step of the message delivery process.

To confirm this cause of the problem: If what appears to be slow connection throughput cannot be explained by any of the causes above, see Factors Affecting Performance for other possible bottlenecks and check for symptoms associated with the following problems:

- Message Production Is Delayed or Slowed

- Messages Are Backlogged

- Broker Throughput Is Sporadic

To resolve the problem: Follow the problem resolution guidelines provided in the troubleshooting sections listed above.

A Client Cannot Create a Message Producer

Symptom:

- A message producer cannot be created for a physical destination; the client receives an exception.

Possible causes:

- A physical destination has been configured to allow only a limited number of producers.

- The user is not authorized to create a message producer due to settings in the access control file.

A physical destination has been configured to allow only a limited number of producers.

One of the ways of avoiding the accumulation of messages on a physical destination is to limit the number of producers (`maxNumProducers`) that it supports.

To confirm this cause of the problem: Check the physical destination:

```
imqcmd query dst
```

(see Viewing Physical Destination Information). The output will show the current number of producers and the value of `maxNumProducers`. If the two values are the same, the number of producers has reached its configured limit. When a new producer is rejected by the broker, the broker returns the exception

```
ResourceAllocationException [C4088]: A JMS destination limit was reached
```

and makes the following entry in the broker log:

```
[B4183]: Producer can not be added to destination
```

To resolve the problem: Increase the value of the `maxNumProducers` property (see Updating Physical Destination Properties).

The user is not authorized to create a message producer due to settings in the access control file.

To confirm this cause of the problem: When a new producer is rejected by the broker, the broker returns the exception

```
JMSSecurityException [C4076]: Client does not have permission to create
producer on destination
```

and makes the following entries in the broker log:

```
[B2041]: Producer on destination denied[B4051]: Forbidden guest.
```

To resolve the problem: Change the access control properties to allow the user to produce messages (see Authorization Rules for Physical Destinations).

Message Production Is Delayed or Slowed

Symptoms:

- When sending persistent messages, the `send` method does not return and the client blocks.

- When sending a persistent message, the client receives an exception.

- A producing client slows down.

Possible causes:

- The broker is backlogged and has responded by slowing message producers.

- The broker cannot save a persistent message to the data store.

- Broker acknowledgment timeout is too short.

- A producing client is encountering JVM limitations.

The broker is backlogged and has responded by slowing message producers.

A backlogged broker accumulates messages in broker memory. When the number of messages or message bytes in physical destination memory reaches configured limits, the broker attempts to conserve memory resources in accordance with the specified limit behavior. The following limit behaviors slow down message producers:

- `FLOW_CONTROL`: The broker does not immediately acknowledge receipt of persistent messages (thereby blocking a producing client).

- `REJECT_NEWEST`: The broker rejects new persistent messages.

Similarly, when the number of messages or message bytes in brokerwide memory (for all physical destinations) reaches configured limits, the broker will attempt to conserve memory resources by rejecting the newest messages. Also, when system memory limits are reached because physical destination or brokerwide limits have not been set properly, the broker takes increasingly serious action to prevent memory overload. These actions include throttling back message producers.

To confirm this cause of the problem: When a message is rejected by the broker because of configured message limits, the broker returns the exception

```
JMSException [C4036]: A server error occurred
```

and makes the following entry in the broker log:

```
[B2011]: Storing of JMS message from IMQconn failed
```

This message is followed by another indicating the limit that has been reached:

```
[B4120]: Cannot store message on destination destName because capacity of
maxNumMsgs would be exceeded.
```

if the exceeded message limit is on a physical destination, or

```
[B4024]: The maximum number of messages currrently in the system has been
exceeded, rejecting message.
```

if the limit is brokerwide.

More generally, you can check for message limit conditions before the rejections occur as follows:

- Query physical destinations and the broker and inspect their configured message limit settings.

- Monitor the number of messages or message bytes currently in a physical destination or in the broker as a whole, using the appropriate `imqcmd` commands. See Metrics Information Reference for information about metrics you can monitor and the commands you use to obtain them.

To resolve the problem:

- Modify the message limits on a physical destination (or brokerwide), being careful not to exceed memory resources.

  In general, you should manage memory at the individual destination level, so that brokerwide message limits are never reached. For more information, see Broker Memory Management Adjustments.

- Change the limit behaviors on a destination so as not to slow message production when message limits are reached, but rather to discard messages in memory.

  For example, you can specify the `REMOVE_OLDEST` and `REMOVE_LOW_PRIORITY` limit behaviors, which delete messages that accumulate in memory (see Table 18–1).

The broker cannot save a persistent message to the data store.

If the broker cannot access a data store or write a persistent message to it, the producing client is blocked. This condition can also occur if destination or brokerwide message limits are reached, as described above.

To confirm this cause of the problem: If the broker is unable to write to the data store, it makes one of the following entries in the broker log:

```
[B2011]: Storing of JMS message from connectionID failed[B4004]: Failed to
persist message messageID
```

To resolve the problem:

- In the case of file-based persistence, try increasing the disk space of the file-based data store.

- In the case of a JDBC-compliant data store, check that JDBC-based persistence is properly configured (seeConfiguring a JDBC-Based Data Store). If so, consult your database administrator to troubleshoot other database problems.

Broker acknowledgment timeout is too short.

Because of slow connections or a lethargic broker (caused by high CPU utilization or scarce memory resources), a broker may require more time to acknowledge receipt of a persistent message than allowed by the value of the connection factory's `imqAckTimeout` attribute.

To confirm this cause of the problem: If the `imqAckTimeout` value is exceeded, the broker returns the exception

```
JMSException [C4000]: Packet acknowledge failed
```

To resolve the problem: Change the value of the `imqAckTimeout` connection factory attribute (see Reliability And Flow Control).

A producing client is encountering JVM limitations.

To confirm this cause of the problem:

- Find out whether the client application receives an out-of-memory error.

- Check the free memory available in the JVM heap, using runtime methods such as `freeMemory`, `maxMemory`, and `totalMemory`.

To resolve the problem: Adjust the JVM (see Java Virtual Machine Adjustments).

Messages Are Backlogged

Symptoms:

- Message production is delayed or produced messages are rejected by the broker.

- Messages take an unusually long time to reach consumers.

- The number of messages or message bytes in the broker (or in specific destinations) increases steadily over time.

To see whether messages are accumulating, check how the number of messages or message bytes in the broker changes over time and compare to configured limits. First check the configured limits:

```
imqcmd query bkr
```

> **Note:** The `imqcmd metrics bkr` subcommand does not display this information.

Then check for message accumulation in each destination:

```
imqcmd list dst
```

To see whether messages have exceeded configured destination or brokerwide limits, check the broker log for the entry

```
[B2011]: Storing of JMS message from ...failed.
```

This entry will be followed by another identifying the limit that has been exceeded.

Possible causes:

- There are inactive durable subscriptions on a topic destination.

- Too few consumers are available to consume messages in a multiple-consumer queue.

- Message consumers are processing too slowly to keep up with message producers.

- Client acknowledgment processing is slowing down message consumption.

- The broker cannot keep up with produced messages.

- Client code defects; consumers are not acknowledging messages.

There are inactive durable subscriptions on a topic destination.

If a durable subscription is inactive, messages are stored in a destination until the corresponding consumer becomes active and can consume the messages.

To confirm this cause of the problem: Check the state of durable subscriptions on each topic destination:

```
imqcmd list dur -d destName
```

To resolve the problem:

- Purge all messages for the offending durable subscriptions (see Managing Durable Subscriptions).

- Specify message limit and limit behavior attributes for the topic (see Table 18–1). For example, you can specify the REMOVE_OLDEST and REMOVE_LOW_PRIORITY limit behaviors, which delete messages that accumulate in memory.

- Purge all messages from the corresponding destinations (see Purging a Physical Destination).

- Limit the time messages can remain in memory by rewriting the producing client to set a time-to-live value on each message. You can override any such settings for all producers sharing a connection by setting the imqOverrideJMSExpiration and imqJMSExpiration connection factory attributes (see Message Header Overrides).

Too few consumers are available to consume messages in a multiple-consumer queue.

If there are too few active consumers to which messages can be delivered, a queue destination can become backlogged as messages accumulate. This condition can occur for any of the following reasons:

- Too few active consumers exist for the destination.

- Consuming clients have failed to establish connections.

- No active consumers use a selector that matches messages in the queue.

To confirm this cause of the problem: To help determine the reason for unavailable consumers, check the number of active consumers on a destination:

```
imqcmd metrics dst -n destName-t q -m con
```

To resolve the problem: Depending on the reason for unavailable consumers,

- Create more active consumers for the queue by starting up additional consuming clients.

- Adjust the imq.consumerFlowLimit broker property to optimize queue delivery to multiple consumers (see Adjusting Multiple-Consumer Queue Delivery).

- Specify message limit and limit behavior attributes for the queue (see Table 18–1). For example, you can specify the REMOVE_OLDEST and REMOVE_LOW_PRIOROTY limit behaviors, which delete messages that accumulate in memory.

- Purge all messages from the corresponding destinations (see Purging a Physical Destination).

- Limit the time messages can remain in memory by rewriting the producing client to set a time-to-live value on each message. You can override any such setting for all producers sharing a connection by setting the imqOverrideJMSExpiration and imqJMSExpiration connection factory attributes (see Message Header Overrides).

Message consumers are processing too slowly to keep up with message producers.

In this case, topic subscribers or queue receivers are consuming messages more slowly than the producers are sending messages. One or more destinations are getting backlogged with messages because of this imbalance.

To confirm this cause of the problem: Check for the rate of flow of messages into and out of the broker:

```
imqcmd metrics bkr -m rts
```

Then check flow rates for each of the individual destinations:

```
imqcmd metrics bkr -t destType-n destName-m rts
```

To resolve the problem:

- Optimize consuming client code.

- For queue destinations, increase the number of active consumers (see Adjusting Multiple-Consumer Queue Delivery).

Client acknowledgment processing is slowing down message consumption.
Two factors affect the processing of client acknowledgments:

- Significant broker resources can be consumed in processing client acknowledgments. As a result, message consumption may be slowed in those acknowledgment modes in which consuming clients block until the broker confirms client acknowledgments.

- JMS payload messages and Message Queue control messages (such as client acknowledgments) share the same connection. As a result, control messages can be held up by JMS payload messages, slowing message consumption.

To confirm this cause of the problem:

- Check the flow of messages relative to the flow of packets. If the number of packets per second is out of proportion to the number of messages, client acknowledgments may be a problem.

- Check to see whether the client has received the following exception:

  ```
  JMSException [C4000]: Packet acknowledge failed
  ```

To resolve the problem:

- Modify the acknowledgment mode used by clients: for example, switch to `DUPS_OK_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE`.

- If using `CLIENT_ACKNOWLEDGE` or transacted sessions, group a larger number of messages into a single acknowledgment.

- Adjust consumer and connection flow control parameters (see Client Runtime Message Flow Adjustments).

The broker cannot keep up with produced messages.
In this case, messages are flowing into the broker faster than the broker can route and dispatch them to consumers. The sluggishness of the broker can be due to limitations in any or all of the following:

- CPU

- Network socket read/write operations

- Disk read/write operations

- Memory paging

- Persistent store

- JVM memory limits

To confirm this cause of the problem: Check that none of the other possible causes of this problem are responsible.

To resolve the problem:

- Upgrade the speed of your computer or data store.

- Use a broker cluster to distribute the load among multiple broker instances.

Client code defects; consumers are not acknowledging messages.

Messages are held in a destination until they have been acknowledged by all consumers to which they have been sent. If a client is not acknowledging consumed messages, the messages accumulate in the destination without being deleted.

For example, client code might have the following defects:

- Consumers using the CLIENT_ACKNOWLEDGE acknowledgment mode or transacted session may not be calling Session.acknowledge or Session.commit regularly.

- Consumers using the AUTO_ACKNOWLEDGE acknowledgment mode may be hanging for some reason.

To confirm this cause of the problem: First check all other possible causes listed in this section. Next, list the destination with the following command:

imqcmd list dst

Notice whether the number of messages listed under the UnAcked header is the same as the number of messages in the destination. Messages under this header were sent to consumers but not acknowledged. If this number is the same as the total number of messages, then the broker has sent all the messages and is waiting for acknowledgment.

To resolve the problem: Request the help of application developers in debugging this problem.

Broker Throughput Is Sporadic

Symptom:

- Message throughput sporadically drops and then resumes normal performance.

- Message throughput sporadically drops and then broker exits.

Possible causes:

- The broker is very low on memory resources.

- JVM memory reclamation (garbage collection) is taking place.

- The JVM is using the just-in-time compiler to speed up performance.

- The Broker exits with an out of memory exception.

The broker is very low on memory resources.

Because destination and broker limits were not properly set, the broker takes increasingly serious action to prevent memory overload; this can cause the broker to become sluggish until the message backlog is cleared.

To confirm this cause of the problem: Check the broker log for a low memory condition

[B1089]: In low memory condition, broker is attempting to free up resources

followed by an entry describing the new memory state and the amount of total memory being used. Also check the free memory available in the JVM heap:

imqcmd metrics bkr -m cxn

Free memory is low when the value of total JVM memory is close to the maximum JVM memory value.

To resolve the problem:

- Adjust the JVM (see Java Virtual Machine Adjustments).

- Increase system swap space.

JVM memory reclamation (garbage collection) is taking place.

Memory reclamation periodically sweeps through the system to free up memory. When this occurs, all threads are blocked. The larger the amount of memory to be freed up and the larger the JVM heap size, the longer the delay due to memory reclamation.

To confirm this cause of the problem: Monitor CPU usage on your computer. CPU usage drops when memory reclamation is taking place.

Also start your broker using the following command line options:

```
-vmargs-verbose:gc
```

Standard output indicates the time when memory reclamation takes place.

To resolve the problem: In computers with multiple CPUs, set the memory reclamation to take place in parallel:

```
-XX:+UseParallelGC=true
```

The JVM is using the just-in-time compiler to speed up performance.

To confirm this cause of the problem: Check that none of the other possible causes of this problem are responsible.

To resolve the problem: Let the system run for awhile; performance should improve.

The Broker exits with an out of memory exception.

To confirm this cause of the problem: Verify the log trace includes `java.lang.OutOfMemoryError: GC overhead limit exceeded`. For example:

```
WARNING: WARNING GC overhead limit exceeded:
java.lang.OutOfMemoryError: GC overhead limit exceeded
at java.util.Arrays.copyOf(Arrays.java:2219)
at java.util.ArrayList.toArray(ArrayList.java:329)
at java.util.Collections$SynchronizedCollection.toArray(Collections.java:1623)
at java.util.ArrayList.<init>(ArrayList.java:151)
at
com.sun.messaging.jmq.jmsserver.data.TransactionReaper.clearSwipeMark(TransactionL
ist.java:3454)
at
com.sun.messaging.jmq.jmsserver.data.TransactionReaper.run(TransactionList.java:35
12)
at com.sun.messaging.jmq.util.timer.WakeupableTimer.run(WakeupableTimer.java:114)
at java.lang.Thread.run(Thread.java:722)
```

To resolve the problem: Set the following JVM GC flag:

```
-XX:+UseConcMarkSweepGC
```

Messages Are Not Reaching Consumers

Symptom:

- Messages sent by producers are not received by consumers.

Possible causes:

- Limit behaviors are causing messages to be deleted on the broker.

- Message timeout value is expiring.

- The broker clock and producer clock are not synchronized.

- Consuming client failed to start message delivery on a connection.

Limit behaviors are causing messages to be deleted on the broker.

When the number of messages or message bytes in destination memory reach configured limits, the broker attempts to conserve memory resources. Three of the configurable behaviors adopted by the broker when these limits are reached will cause messages to be lost:

- `REMOVE_OLDEST`: Delete the oldest messages.

- `REMOVE_LOW_PRIORITY`: Delete the lowest-priority messages according to age.

- `REJECT_NEWEST`: Reject new persistent messages.

To confirm this cause of the problem: Use the QBrowser demo application to inspect the contents of the dead message queue (see To Inspect the Dead Message Queue).

Check whether the `JMS_SUN_DMQ_UNDELIVERED_REASON` property of messages in the queue has the value `REMOVE_OLDEST` or `REMOVE_LOW_PRIORITY`.

To resolve the problem: Increase the destination limits. For example:

```
imqcmd update dst -n MyDest-o maxNumMsgs=1000
```

Message timeout value is expiring.

The broker deletes messages whose timeout value has expired. If a destination gets sufficiently backlogged with messages, messages whose time-to-live value is too short might be deleted.

To confirm this cause of the problem: Use the QBrowser demo application to inspect the contents of the dead message queue (see To Inspect the Dead Message Queue).

Check whether the `JMS_SUN_DMQ_UNDELIVERED_REASON` property of messages in the queue has the value `EXPIRED`.

To resolve the problem: Contact the application developers and have them increase the time-to-live value.

The broker clock and producer clock are not synchronized.

If clocks are not synchronized, broker calculations of message lifetimes can be wrong, causing messages to exceed their expiration times and be deleted.

To confirm this cause of the problem: Use the QBrowser demo application to inspect the contents of the dead message queue (see To Inspect the Dead Message Queue).

Check whether the `JMS_SUN_DMQ_UNDELIVERED_REASON` property of messages in the queue has the value `EXPIRED`.

In the broker log file, look for any of the following messages: `B2102`, `B2103`, `B2104`. These messages all report that possible clock skew was detected.

To resolve the problem: Check that you are running a time synchronization program, as described in Preparing System Resources.

Consuming client failed to start message delivery on a connection.
Messages cannot be delivered until client code establishes a connection and starts message delivery on the connection.

To confirm this cause of the problem: Check that client code establishes a connection and starts message delivery.

To resolve the problem: Rewrite the client code to establish a connection and start message delivery.

Dead Message Queue Contains Messages

Symptom:

- When you list destinations, you see that the dead message queue contains messages. For example, issue a command like the following:

  ```
  imqcmd list dst
  ```

  After you supply a user name and password, output like the following appears:

```
Listing all the destinations on the broker specified by:
---------------------------------
Host         Primary Port
---------------------------------
localhost    7676

----------------------------------------------------------------------
   Name     Type     State    Producers  Consumers  Msgs
                                          Total      Count  UnAck  InDelay Avg Size
----------------------------------------------------  ----------------------------
MyDest      Queue    RUNNING      0          0          5      0      0     1177.0
mq.sys.dmq  Queue    RUNNING      0          0         35      0      0     1422.0
Successfully listed destinations.
```

In this example, the dead message queue, mq.sys.dmq, contains 35 messages.

Possible causes:

- The number of messages, or their sizes, exceed destination limits..

- The broker clock and producer clock are not synchronized..

- An unexpected broker error has occurred..

- Consumers are not consuming messages before they time out..

  There are a number of possible reasons for messages to time out:

  - There are too many producers for the number of consumers..

  - Producers are faster than consumers..

  - A consumer is too slow..

  - Clients are not committing transactions..

  - Consumers are failing to acknowledge messages..

  - Durable subscribers are inactive..

The number of messages, or their sizes, exceed destination limits.

To confirm this cause of the problem: Use the QBrowser demo application to inspect the contents of the dead message queue (see To Inspect the Dead Message Queue).

Check the values for the following message properties:

- `JMS_SUN_DMQ_UNDELIVERED_REASON`

- `JMS_SUN_DMQ_UNDELIVERED_COMMENT`

- `JMS_SUN_DMQ_UNDELIVERED_TIMESTAMP`

Under `JMS Headers`, scroll down to the value for `JMSDestination` to determine the destination whose messages are becoming dead.

To resolve the problem: Increase the destination limits. For example:

`imqcmd update dst -n MyDest -o maxNumMsgs=1000`

The broker clock and producer clock are not synchronized.

If clocks are not synchronized, broker calculations of message lifetimes can be wrong, causing messages to exceed their expiration times and be deleted.

To confirm this cause of the problem: Use the QBrowser demo application to inspect the contents of the dead message queue (see To Inspect the Dead Message Queue).

Check whether the `JMS_SUN_DMQ_UNDELIVERED_REASON` property of messages in the queue has the value `EXPIRED`.

In the broker log file, look for any of the following messages: `B2102`, `B2103`, `B2104`. These messages all report that possible clock skew was detected.

To resolve the problem: Check that you are running a time synchronization program, as described in Preparing System Resources.

An unexpected broker error has occurred.

To confirm this cause of the problem: Use the QBrowser demo application to inspect the contents of the dead message queue (see To Inspect the Dead Message Queue).

Check whether the `JMS_SUN_DMQ_UNDELIVERED_REASON` property of messages in the queue has the value `ERROR`.

To resolve the problem:

- Examine the broker log file to find the associated error.

- Contact Oracle Technical Support to report the broker problem.

Consumers are not consuming messages before they time out.

To confirm this cause of the problem: Use the QBrowser demo application to inspect the contents of the dead message queue (see To Inspect the Dead Message Queue).

Check whether the `JMS_SUN_DMQ_UNDELIVERED_REASON` property of messages in the queue has the value `EXPIRED`.

Check to see if there any consumers on the destination and the value for the `Current Number of Active Consumers`. For example:

`imqcmd query dst -t q -n MyDest`

If there are active consumers, then there might be any number of possible reasons why messages are timing out before being consumed. One is that the message timeout is too short for the speed at which the consumer executes. In that case, request that application developers increase message time-to-live values. Otherwise, investigate the following possible causes for messages to time out before being consumed:

There are too many producers for the number of consumers.

To confirm this cause of the problem: Use the QBrowser demo application to inspect the contents of the dead message queue (see To Inspect the Dead Message Queue).

Check whether the `JMS_SUN_DMQ_UNDELIVERED_REASON` property of messages in the queue has the value `REMOVE_OLDEST` or `REMOVE_LOW_PRIORITY`. If so, use the `imqcmd query dst` command to check the number of producers and consumers on the destination. If the number of producers exceeds the number of consumers, the production rate might be overwhelming the consumption rate.

To resolve the problem: Add more consumer clients or set the destination's limit behavior to `FLOW_CONTROL` (which uses consumption rate to control production rate), using a command such as the following:

```
imqcmd update dst -n myDst -t q -o limitBehavior=FLOW_CONTROL
```

Producers are faster than consumers.

To confirm this cause of the problem: To determine whether slow consumers are causing producers to slow down, set the destination's limit behavior to `FLOW_CONTROL` (which uses consumption rate to control production rate), using a command such as the following:

```
imqcmd update dst -n myDst -t q -o limitBehavior=FLOW_CONTROL
```

Use metrics to examine the destination's input and output, using a command such as the following:

```
imqcmd metrics dst -n myDst -t q -m rts
```

In the metrics output, examine the following values:

- `Msgs/sec Out`: Shows how many messages per second the broker is removing. The broker removes messages when all consumers acknowledge receiving them, so the metric reflects consumption rate.

- `Msgs/sec In`: Shows how many messages per second the broker is receiving from producers. The metric reflects production rate.

Because flow control aligns production to consumption, note whether production slows or stops. If so, there is a discrepancy between the processing speeds of producers and consumers. You can also check the number of unacknowledged (`UnAcked`) messages sent, by using the `imqcmd list dst` command. If the number of unacknowledged messages is less than the size of the destination, the destination has additional capacity and is being held back by client flow control.

To resolve the problem: If production rate is consistently faster than consumption rate, consider using flow control regularly, to keep the system aligned. In addition, consider and attempt to resolve each of the following possible causes, which are subsequently described in more detail:

- A consumer is too slow..

- Clients are not committing transactions..

- Consumers are failing to acknowledge messages..

- Durable subscribers are inactive..

- An unexpected broker error has occurred..

A consumer is too slow.

To confirm this cause of the problem: Use `imqcmd metrics` to determine the rate of production and consumption, as described above under Producers are faster than consumers..

To resolve the problem:

- Set the destinations' limit behavior to `FLOW_CONTROL`, using a command such as the following:

  ```
  imqcmd update dst -n myDst -t q -o limitBehaviort=FLOW_CONTROL
  ```

  Use of flow control slows production to the rate of consumption and prevents the accumulation of messages in the destination. Producer applications hold messages until the destination can process them, with less risk of expiration.

- Find out from application developers whether producers send messages at a steady rate or in periodic bursts. If an application sends bursts of messages, increase destination limits as described in the next item.

- Increase destination limits based on number of messages or bytes, or both. To change the number of messages on a destination, enter a command with the following format:

  ```
  imqcmd update dst -n destName-t {q|t} -o maxNumMsgs=number
  ```

  To change the size of a destination, enter a command with the following format:

  ```
  imqcmd update dst -n destName-t {q|t} -o maxTotalMsgBytes=number
  ```

  Be aware that raising limits increases the amount of memory that the broker uses. If limits are too high, the broker could run out of memory and become unable to process messages.

- Consider whether you can accept loss of messages during periods of high production load.

Clients are not committing transactions.

To confirm this cause of the problem: Check with application developers to find out whether the application uses transactions. If so, list the active transactions as follows:

```
imqcmd list txn
```

Here is an example of the command output:

```
----------------------------------------------------------------------
Transaction ID       State    User name   # Msgs/# Acks   Creation time
----------------------------------------------------------------------
6800151593984248832  STARTED  guest            3/2        7/19/04 11:03:08 AM
```

Note the numbers of messages and number of acknowledgments. If the number of messages is high, producers may be sending individual messages but failing to commit transactions. Until the broker receives a commit, it cannot route and deliver the messages for that transaction. If the number of acknowledgments is high, consumers may be sending acknowledgments for individual messages but failing to commit transactions. Until the broker receives a commit, it cannot remove the acknowledgments for that transaction.

To resolve the problem: Contact application developers to fix the coding error.

Consumers are failing to acknowledge messages.

To confirm this cause of the problem: Contact application developers to determine whether the application uses system-based acknowledgment (AUTO_ACKNOWLEDGE or DUPES_ONLY) or client-based acknowledgment (CLIENT_ACKNOWLEDGE). If the application uses system-based acknowledgment , skip this section; if it uses client-based acknowledgment), first decrease the number of messages stored on the client, using a command like the following:

```
imqcmd update dst -n myDst -t q -o consumerFlowLimit=1
```

Next, you will determine whether the broker is buffering messages because a consumer is slow, or whether the consumer processes messages quickly but does not acknowledge them. List the destination, using the following command:

```
imqcmd list dst
```

After you supply a user name and password, output like the following appears:

```
Listing all the destinations on the broker specified by:
-------------------------------
Host        Primary Port
-------------------------------
localhost   7676

----------------------------------------------------------------------
   Name       Type     State   Producers  Consumers  Msgs
                                          Total      Count  UnAck  InDelay  Avg Size
-------------------------------------------------- --------------------------
MyDest      Queue   RUNNING     0          0          5    200       0  1177.0
mq.sys.dmq  Queue   RUNNING     0          0         35      0       0  1422.0
Successfully listed destinations.
```

The UnAck number represents messages that the broker has sent and for which it is waiting for acknowledgment. If this number is high or increasing, you know that the broker is sending messages, so it is not waiting for a slow consumer. You also know that the consumer is not acknowledging the messages.

To resolve the problem: Contact application developers to fix the coding error.

Durable subscribers are inactive.

To confirm this cause of the problem: Look at the topic's durable subscribers, using the following command format:

```
imqcmd list dur -d topicName
```

To resolve the problem:

- Purge the durable subscribers using the `imqcmd purge dur` command.
- Restart the consumer applications.

To Inspect the Dead Message Queue

A number of troubleshooting procedures involve an inspection of the dead message queue (mq.sys.dmq). The following procedure explains how to carry out such an inspection by using the QBrowser demo application.
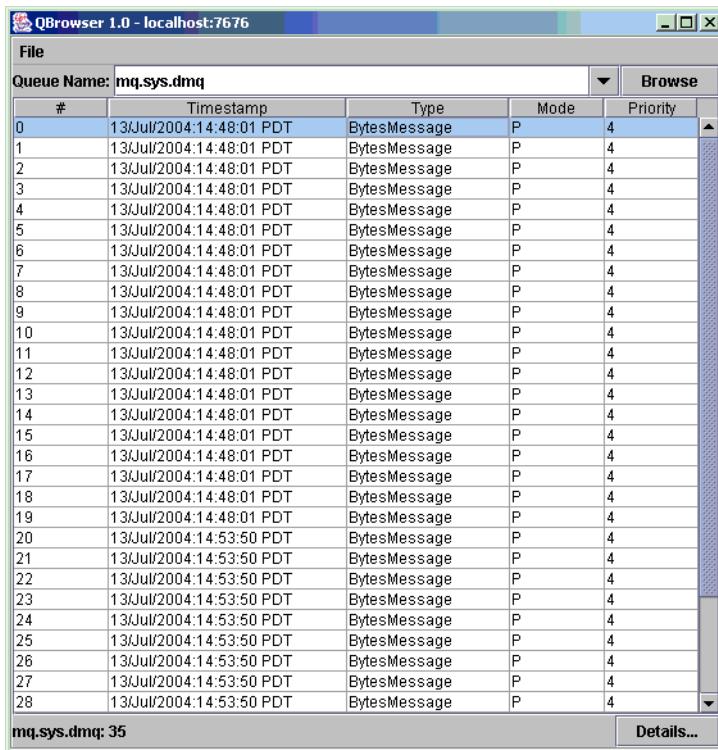
1. Locate the QBrowser demo application in IMQ_HOME/examples/applications/qbrowser.

2. Run the QBrowser application:

```
java QBrowser
```

The QBrowser main window appears.

3. Select the queue name `mq.sys.dmq` and click Browse.

A list like the following appears:

4. Double-click any message to display details about that message:

The display should resemble the following:

You can inspect the Message Properties pane to determine the reason why the message was placed in the dead message queue.

Part III

Reference

16

Command Line Reference

This chapter provides reference information on the use of the Message Queue command line administration utilities. It consists of the following sections:

- Command Line Syntax
- Broker Utility
- Command Utility
- Object Manager Utility
- Database Manager Utility
- User Manager Utility
- Bridge Manager Utility
- Service Administrator Utility
- Key Tool Utility

Command Line Syntax

Message Queue command line utilities are shell commands. The name of the utility is a command and its subcommands or options are arguments passed to that command. There is no need for separate commands to start or quit the utility.

All the command line utilities share the following command syntax:

utilityName [*subcommand*] [*commandArgument*] [*-optionName* [*optionArgument*]]...

where *utilityName* is one of the following:

- `imqbrokerd` (Broker utility)
- `imqcmd` (Command utility)
- `imqobjmgr` (Object Manager utility)
- `imqdbmgr` (Database Manager utility)
- `imqusermgr` (User Manager utility)
- `imqbridgemgr` (Bridge Manager utility)
- `imqsvcadmin` (Service Administrator utility)
- `imqkeytool` (Key Tool utility)

Subcommands and command-level arguments, if any, must precede all options and their arguments; the options themselves may appear in any order. All subcommands,

command arguments, options, and option arguments are separated with spaces. If the value of an option argument contains a space, the entire value must be enclosed in quotation marks. (It is generally safest to enclose any attribute-value pair in quotation marks.)

The following command, which starts the default broker, is an example of a command line with no subcommand clause:

```
imqbrokerd
```

Here is a fuller example:

```
imqcmddestroy dst-t q-n myQueue-u admin-f-s
```

This command destroys a queue destination (destination type `q`) named `myQueue`. Authentication is performed on the user name `admin`; the command will prompt for a password. The command will be performed without prompting for confirmation (`-f` option) and in silent mode, without displaying any output (`-s` option).

Broker Utility

The Broker utility (`imqbrokerd`) starts a broker. Command line options override values in the broker configuration files, but only for the current broker session.

Table 16–1 shows the options to the `imqbrokerd` command and the configuration properties, if any, overridden by each option.

Table 16–1   Broker Utility Options

Option	Description
`-name` *instanceName*	Instance name of broker
	Multiple broker instances running on the same host must have different instance names.
	Default value: `imqbroker`
	Properties overridden: `imq.instancename`
`-port` *portNumber*	Port number for broker's Port Mapper
	Message Queue clients use this port number to connect to the broker. Multiple broker instances running on the same host must have different Port Mapper port numbers.
	Default value: `7676`
	Properties overridden: `imq.portmapper.port`
`-cluster` *broker1* [`,`*broker2*]`...`	Connect brokers into cluster[1]
	The specified brokers are merged with the list in the `imq.cluster.brokerlist` property. Each broker argument has one of the forms
	■ *hostName*`:`*portNumber*
	■ *hostName*
	■ `:`*portNumber*
	If *hostName* is omitted, the default value is `localhost`; if *portNumber* is omitted, the default value is `7676`.
	Literal IP addresses as host names: You can use a literal IPv4 or IPv6 address as a host name. If you use a literal IPv6 address, its format must conform to RFC2732 (http://www.ietf.org/rfc/rfc2732.txt), *Format for Literal IPv6 Addresses in URL's*.
	Properties overridden: `imq.cluster.brokerlist`

Table 16–1   (Cont.) Broker Utility Options

Option	Description
-Dproperty=value	Set configuration property
	See Broker Properties Reference for information about broker configuration properties.
	Caution: Be careful to check the spelling and formatting of properties set with this option. Incorrect values will be ignored without notification or warning.
	Properties overridden: Corresponding property in instance configuration file
-reset props	Reset configuration properties
	Replaces the broker's existing instance configuration file config.properties with an empty file; all properties assume their default values.
	Properties overridden: None
-reset store	Reset persistent data store
	Clears all persistent data from the data store (including persistent messages, durable subscriptions, and transaction information), allowing you to start the broker instance with a clean slate. To prevent the persistent store from being reset on subsequent restarts, restart the broker instance without the -reset option.
	To clear only persistent messages or durable subscriptions, use -reset messages or -reset durables instead.
	Properties overridden: None
-reset messages	Clear persistent messages from data store
	Properties overridden: None
-reset durables	Clear durable subscriptions from data store
	Properties overridden: None
-reset takeover-then-exit	Clear any takeover locks and then exit.
	If a broker fails before completing the takeover of another broker's store, the failed broker retains a takeover lock even though it is not running. Use -reset takeover-then-exit to cause the failed broker to initiate startup, release the takeover lock, and then exit without actually starting up.
	Properties overridden: None
-backup fileName	Back up configuration change record to file[1]
	See Managing a Conventional Cluster's Configuration Change Record for more information.
	Properties overridden: None
-restore fileName	Restore configuration change record from backup file[1]
	The backup file must have been previously created using the -backup option.
	See Managing a Conventional Cluster's Configuration Change Record for more information.
	Properties overridden: None
-remove instance	Remove broker instance[2]
	Deletes the instance configuration file, log files, persistent store, and other files and directories associated with the instance.
	Properties overridden: None
-dbuser userName	User name for JDBC-based persistent data store
	Properties overridden: imq.persist.jdbc.user

Table 16–1   (Cont.) Broker Utility Options

Option	Description
-passfile *filePath*	Location of password file
	Sets the broker's `imq.passfile.enabled` property to `true`, `imq.passfile.dirpath` to the path containing the password file, and `imq.passfile.name` to the file name itself.
	See Password Files for more information.
	Properties overridden: `imq.passfile.enabledimq.passfile.dirpathimq.passfile.name`
-shared	Use shared thread pool model to implement `jms` connection service
	Execution threads will be shared among connections to increase the number of connections supported.
	Sets the broker's `imq.jms.threadpool_model` property to `shared`.
	Properties overridden: `imq.jms.threadpool_model`
-javahome *path*	Location of alternative Java runtime
	Default behavior: Use runtime installed on system or bundled with Message Queue.
	Properties overridden: None
-vmargs *arg1* [*arg2*] ...	Pass arguments to Java virtual machine
	Arguments are separated with spaces. To pass more than one argument, or an argument containing a space, enclose the argument list in quotation marks.
	VM arguments can be passed only from the command line; there is no associated configuration property in the instance configuration file.
	Properties overridden: None
-startRmiRegistry	Start RMI registry at broker startup
	Properties overridden: `imq.jmx.rmiregistry.start`
-useRmiRegistry	Use external RMI registry
	Properties overridden: `imq.jmx.rmiregistry.use`
-rmiRegistryPort	Port number of RMI registry
	Properties overridden: `imq.jmx.rmiregistry.port`
-upgrade-store-nobackup	Automatically remove old data store on upgrade to Message Queue 3.5 or 3.5 SP*x* from an incompatible version[2]
	Properties overridden: None
-force	Perform action without user confirmation
	This option applies only to the `-remove instance` and `-upgrade-store-nobackup` options, which normally require confirmation.
	Properties overridden: None
-loglevel *level*	Logging level: `NONE`, `ERROR`, `WARNING` or `INFO`.
	Default value: `INFO`
	Properties overridden: `imq.broker.log.level`
-metrics *interval*	Logging interval for broker metrics, in seconds
	Properties overridden: `imq.metrics.interval`

Table 16–1   (Cont.) Broker Utility Options

Option	Description
-tty	Log all messages to console
	Sets the broker's `imq.log.console.output` property to `ALL`.
	If not specified, only error and warning messages will be logged.
	Properties overridden: `imq.log.console.output`
-s | -silent	Silent mode (no logging to console)
	Sets the broker's `imq.log.console.output` property to `NONE`.
	Properties overridden: `imq.log.console.output`
-version	Display version information[3]
	Properties overridden: None
-h | -help	Display usage help[3]
	Properties overridden: None

[1]   Applies only to broker clusters
[2]   Requires user confirmation unless `-force` is also specified
[3]   Any other options specified on the command line are ignored.

Command Utility

The Command utility (`imqcmd`) is used for managing brokers, connection services, connections, physical destinations, durable subscriptions, and transactions.

All `imqcmd` commands must include a subcommand (except those using the `-v` or `-h` option to display product version information or usage help, respectively). The possible subcommands are listed in Table 16–2 and described in detail in the corresponding sections below. In addition, each `imqcmd` subcommand supports the general options shown in General Command Utility Options.

> **Note:**   The `-u` *userName* option (and corresponding password) is required except when using the `-v` or `-h` option. Also if a subcommand accepts a broker address (`-b` option) and no host name or port number is specified, the values `localhost` and `7676` are assumed by default.

Table 16–2   Command Utility Subcommands

Command	Description
Broker Management	
shutdown bkr	Shut down broker
restart bkr	Restart broker
pause bkr	Pause broker
quiesce bkr	Quiesce broker
unquiesce bkr	Unquiesce broker
resume bkr	Resume broker
takeover bkr	Initiate broker takeover
update bkr	Set broker properties

Table 16–2   (Cont.) Command Utility Subcommands

Command	Description
query bkr	List broker property values
list bkr	List brokers in cluster
metrics bkr	Display broker metrics
reload cls	Reload cluster configuration
changemaster cls	Change the master broker in a conventional cluster with master broker
Connection Service Management	
pause svc	Pause connection service
resume svc	Resume connection service
update svc	Set connection service properties
list svc	List connection services available on broker
query svc	List connection service property values
metrics svc	Display connection service metrics
Connection Management	
list cxn	List connections on broker
query cxn	Display connection information
destroy cxn	Destroy connection
Physical Destination Management	
create dst	Create physical destination
destroy dst	Destroy physical destination
pause dst	Pause message delivery for physical destination
resume dst	Resume message delivery for physical destination
purge dst	Purge all messages from physical destination
compact dst	Compact physical destination
update dst	Set physical destination properties
list dst	List physical destinations
query dst	List physical destination property values
metrics dst	Display physical destination metrics
Durable Subscription Management	
destroy dur	Destroy durable subscription
purge dur	Purge all messages for durable subscription
list dur	List durable subscriptions for topics
Transaction Management	
commit txn	Commit transaction
rollback txn	Roll back transaction

Table 16–2   (Cont.) Command Utility Subcommands

Command	Description
`list txn`	List transactions being tracked by broker
`query txn`	Display transaction information
`list dur`	List durable subscriptions for topic
JMX Management	
`list jmx`	List JMX service URLs of JMX connectors

General Command Utility Options

The additional options listed in Table 16–3 can be applied to any subcommand of the `imqcmd` command.

Table 16–3   General Command Utility Options

Option	Description
`-secure`	Use secure connection to broker with `ssladmin` connection service
`-u` *userName*	User name for authentication
	If this option is omitted, the Command utility will prompt for it interactively.
`-passfile` *path*	Location of password file
	See Password Files for more information.
`-D`	Set connection-related system property that affects how `imqcmd` creates a connection to the broker. Not used to set broker configuration properties.
	Usually overrides connection factory attributes for `imqcmd` client runtime. For example, the option in the following command changes the default value of `imqSSLIsTrusted`:
	`imqcmd list svc -secure -DimqSSLIsTrusted=true`
`-rtm` *timeoutInterval*	Initial timeout interval, in seconds
	This is the initial length of time that the Command utility will wait for a reply from the broker before retrying a request. Each subsequent retry will use a timeout interval that is a multiple of this initial interval.
	Default value: `10`.
`-rtr` *numRetries*	Number of retries to attempt after a broker request times out
	Default value: `5`.
`-javahome` *path*	Location of alternative Java runtime
	Default behavior: Use runtime installed on system or bundled with Message Queue.
`-f`	Perform action without user confirmation
`-s`	Silent mode (no output displayed)
`-v`	Display version information[1,2]
`-h`	Display usage help[1,2]
`-H`	Display expanded usage help, including attribute list and examples[1,2]

[1] Any other options specified on the command line are ignored.
[2] User name and password not needed

Broker Management

The Command utility cannot be used to start a broker; use the Broker utility (imqbrokerd) instead. Once the broker is started, you can use the imqcmd subcommands listed in Table 16–4 to manage and control it.

Table 16–4   Command Utility Subcommands for Broker Management

Syntax	Description
shutdown bkr [-b *hostName*:*portNumber*] [-time *nSeconds*] [-nofailover]	Shut down broker
	The -time option specifies the interval, in seconds, to wait before shutting down the broker. (The broker will not block, but will return immediately from the delayed shutdown request.) During the shutdown interval, the broker will not accept any new jms connections; admin connections will be accepted, and existing jms connections will continue to operate. A broker belonging to an enhanced cluster will not attempt to take over for any other broker during the shutdown interval.
	The -nofailover option indicates that no other broker is to take over the persistent data of the one being shut down. [1]
restart bkr [-b *hostName*:*portNumber*]	Restart broker
	Shuts down the broker and then restarts it using the same options specified when it was originally started.
pause bkr [-b *hostName*:*portNumber*]	Pause broker
	See Pausing and Resuming a Broker for more information.
quiesce bkr [-b *hostName*:*portNumber*]	Quiesce broker
	The broker will stop accepting new connections; existing connections will continue to operate.
unquiesce bkr [-b *hostName*:*portNumber*]	Unquiesce broker
	The broker will resume accepting new connections, returning to normal operation.
resume bkr [-b *hostName*:*portNumber*]	Resume broker
takeover bkr -n *brokerID* [-f]	Initiate broker takeover [1]
	Before taking over a broker, you should first shut it down manually using the shutdown bkr subcommand with the -nofailover option. If the specified broker appears to be still running, takeover bkr will display a confirmation message (Do you want to take over for this broker?). The -f option suppresses this message and initiates the takeover unconditionally.
	Note: The takeover bkr subcommand is intended only for use in failed-takeover situations. You should use it only as a last resort, and not as a general way of forcibly taking over a running broker.
update bkr [-b *hostName*:*portNumber*] -o *property1*=*value1* [-o *property2*=*value2*]...	Set broker properties
	See Broker Properties Reference for information on broker properties.
query bkr -b *hostName*:*portNumber*	List broker property values
	For brokers belonging to a cluster, also lists cluster properties such as broker list, master broker (for conventional clusters), and cluster identifier (for enhanced clusters).
list bkr	List brokers in cluster

Table 16–4   (Cont.) Command Utility Subcommands for Broker Management

Syntax	Description

`metrics bkr [-b hostName:portNumber]`
`[-m metricType]`
`[-int interval]`
`[-msp numSamples]`
	Display broker metrics
	
	The `-m` option specifies the type of metrics to display:
	
	■ `ttl`: Messages and packets flowing into and out of the broker
	
	■ `rts`: Rate of flow of messages and packets into and out of the broker per second
	
	■ `cxn`: Connections, virtual memory heap, and threads
	
	Default value: `ttl`.
	
	The `-int` option specifies the interval, in seconds, at which to display metrics. **Default value:** 5.
	
	The `-msp` option specifies the number of samples to display. **Default value:** Unlimited (infinite).

`reload cls`	Reload cluster configuration[1] Forces all persistent information to be brought up to date.
`changemaster cls` `-o imq.cluster.masterbroker=newMaster`	Change the master broker in a conventional cluster with master broker. This command must be run on the current master broker. The value *newMaster* has the form *hostName:portNumber*, where *hostName* and *portNumber* are is its Port Mapper host name and port number, respectively. **Literal IP addresses as host names:** You can use a literal IPv4 or IPv6 address as a host name. If you use a literal IPv6 address, its format must conform to RFC2732 (`http://www.ietf.org/rfc/rfc2732.txt`), *Format for Literal IPv6 Addresses in URL's*.

[1]   Applies only to broker clusters

Connection Service Management

Table 16–5 lists the `imqcmd` subcommands for managing connection services.

Table 16–5   Command Utility Subcommands for Connection Service Management

Syntax	Description
`pause svc -n serviceName` `[-bhostName:portNumber]`	Pause connection service The `admin` connection service cannot be paused.
`resume svc -n serviceName` `[-b hostName:portNumber]`	Resume connection service

`update svc -n serviceName`
`[-b hostName:portNumber]`
`-o property1=value1`
`[-o property2=value2]...`
	Set connection service properties
	
	See Connection Properties for information on connection service properties.

Table 16–5   (Cont.) Command Utility Subcommands for Connection Service Management

Syntax	Description
`list svc` `[-b hostName:portNumber]`	List connection services available on broker
`query svc -n serviceName` `[-bhostName:portNumber]`	List connection service property values

`metrics svc -n serviceName`
  `[-bhostName:portNumber]`
  `[-mmetricType]`
  `[-intinterval]`
  `[-mspnumSamples]`
	Display connection service metrics
	
	The `-m` option specifies the type of metrics to display:
	
	■ `ttl`: Messages and packets flowing into and out of the broker by way of the specified connection service
	
	■ `rts`: Rate of flow of messages and packets into and out of the broker per second by way of the specified connection service
	
	■ `cxn`: Connections, virtual memory heap, and threads
	
	Default value: `ttl`.
	
	The `-int` option specifies the interval, in seconds, at which to display metrics. **Default value:** `5`.
	
	The `-msp` option specifies the number of samples to display. **Default value:** Unlimited (infinite).

Connection Management

Table 16–6 lists the `imqcmd` subcommands for managing connections.

Table 16–6   Command Utility Subcommands for Connection Service Management

Syntax	Description
`list cxn [-svn serviceName]` `[-b hostName:portNumber]`	List connections on broker Lists all connections on the broker to the specified connection service. If no connection service is specified, all connections are listed.
`query cxn -n connectionID` `[-b hostName:portNumber]`	Display connection information
`destroy cxn -n connectionID` `[-b hostName:portNumber]`	Destroy connection

Physical Destination Management

Table 16–7 lists the `imqcmd` subcommands for managing physical destinations. In all cases, the `-t` (destination type) option can take either of two values:

■   `q`: Queue destination

■   `t`: Topic destination

Table 16–7   Command Utility Subcommands for Physical Destination Management

Syntax	Description
`create dst -t destType -n destName` `[-o property=value]...`	Create physical destination[1]
	The destination name *destName* may contain only alphanumeric characters (no spaces) and must begin with an alphabetic character or the underscore (_) or dollar sign ($) character. It may not begin with the characters `mq`.
`destroy dst -t destType -n destName`	Destroy physical destination[1]
	This operation cannot be applied to a system-created destination, such as a dead message queue.
`pause dst [-t destType -n destName]` `[-pst pauseType]`	Pause message delivery for physical destination
	Pauses message delivery for the physical destination specified by the `-t` and `-n` options. If these options are not specified, all destinations are paused.
	The `-pst` option specifies the type of message delivery to be paused:
	■ `PRODUCERS`: Pause delivery from message producers
	■ `CONSUMERS`: Pause delivery to message consumers
	■ `ALL`: Pause all message delivery
	Default value: `ALL`
`resume dst [-t destType -n destName]`	Resume message delivery for physical destination
	Resumes message delivery for the physical destination specified by the `-t` and `-n` options. If these options are not specified, all destinations are resumed.
`purge dst -t destType -n destName`	Purge all messages from physical destination
`compact dst [-t destType -n destName]`	Compact physical destination
	Compacts the file-based persistent data store for the physical destination specified by the `-t` and `-n` options. If these options are not specified, all destinations are compacted.
	A destination must be paused before it can be compacted.

`update dst -t destType -n destName`
`-o property1=value1`
`[-o property2=value2]...`
	Set physical destination properties

	See Physical Destination Property Reference for information on physical destination properties.

Table 16–7 (Cont.) Command Utility Subcommands for Physical Destination Management

Syntax	Description
`list dst [-t`*destType*`]` `[-tmp]`	List physical destinations
	Lists all physical destinations of the type specified by the `-t` option. If no destination type is specified, both queue and topic destinations are listed. If the `-tmp` option is specified, temporary destinations are listed as well.
`query dst -t` *destType* `-n` *destName*	List physical destination property values

`metrics dst -t` *destType* `-n` *destName*
`[-m` *metricType*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
	Display physical destination metrics

	The `-m` option specifies the type of metrics to display:
	■ `ttl`: Messages and packets flowing into and out of the destination and residing in memory
	■ `rts`: Rate of flow of messages and packets into and out of the destination per second, along with other rate information
	■ `con`: Metrics related to message consumers
	■ `dsk`: Disk usage
	Default value: `ttl`.
	The `-int` option specifies the interval, in seconds, at which to display metrics. **Default value:** 5.
	The `-msp` option specifies the number of samples to display. **Default value:** Unlimited (infinite).

[1] Cannot be performed in a broker cluster whose master broker is temporarily unavailable

Durable Subscription Management

Table 16–8 lists the `imqcmd` subcommands for managing durable subscriptions.

Table 16–8 Command Utility Subcommands for Durable Subscription Management

Syntax	Description
`destroy dur -n` *subscriberName* `[-c` *clientID*`]`	Destroy durable subscription[1]
`purge dur -n` *subscriberName* `[-c` *clientID*`]`	Purge all messages for durable subscription
`list dur [-d` *topicName*`]`	List durable subscriptions for the specified topic. If `-d` option is omitted then the command lists all durable subscriptions for all topics.

[1] Cannot be performed in a conventional broker cluster whose master broker is temporarily unavailable

Transaction Management

Table 16–9 lists the `imqcmd` subcommands for managing local (non-distributed) Message Queue transactions. Distributed transactions are managed by a distributed transaction manager rather than `imqcmd`.

Table 16–9 Command Utility Subcommands for Transaction Management

Syntax	Description
`commit txn -n` *transactionID*	Commit transaction

Table 16–9   (Cont.) Command Utility Subcommands for Transaction Management

Syntax	Description
rollback txn -n *transactionID*	Roll back transaction
list txn	List transactions being tracked by broker
query txn -n *transactionID*	Display transaction information

JMX Management

The imqcmd subcommand shown in Table 16–10 is used for administrative support of Java applications using the Java Management Extensions (JMX) application programming interface to configure and monitor Message Queue resources. See JMX Support for further information on the broker's JMX support.

Table 16–10   Command Utility Subcommand for JMX Management

Syntax	Description
list jmx	List JMX service URLs of JMX connectors

Object Manager Utility

The Object Manager utility (imqobjmgr) creates and manages Message Queue administered objects. Table 16–11 lists the available subcommands.

Table 16–11   Object Manager Subcommands

Subcommand	Description
add	Add administered object to object store
delete	Delete administered object from object store
list	List administered objects in object store
query	Display administered object information
update	Modify administered object

Table 16–12 lists the options to the imqobjmgr command.

Table 16–12   Object Manager Options

Option	Description
-l *lookupName*	JNDI lookup name of administered object
-j *attribute*=*value*	Attributes of JNDI object store (see Object Stores)
-t *objectType*	Type of administered object: ■ q: Queue destination ■ t: Topic destination ■ cf: Connection factory ■ qf: Queue connection factory ■ tf: Topic connection factory ■ xcf: Connection factory for distributed transactions ■ xqf: Queue connection factory for distributed transactions ■ xtf: Topic connection factory for distributed transactions

Table 16–12 (Cont.) Object Manager Options

Option	Description
`-o` *attribute*=*value*	Attributes of administered object (see Administered Object Attributes and Administered Object Attribute Reference)
`-r` *readOnlyState*	Is administered object read-only? If `true`, client cannot modify object's attributes. **Default value:** `false`.
`-i` *fileName*	Name of command file containing all or part of subcommand clause
`-pre`	Preview results without performing command This option is useful for checking the values of default attributes.
`-javahome` *path*	Location of alternative Java runtime **Default behavior:** Use runtime installed on system or bundled with Message Queue.
`-f`	Perform action without user confirmation
`-s`	Silent mode (no output displayed)
`-v`	Display version information[1]
`-h`	Display usage help[1]
`-H`	Display expanded usage help, including attribute list and examples[1]

[1] Any other options specified on the command line are ignored.

Database Manager Utility

The Database Manager utility (`imqdbmgr`) sets up the database schema for a JDBC-based data store. You can also use it to delete Message Queue database tables that have become corrupted, change the database, display information about the database, convert a standalone database for use in an enhanced broker cluster, or back up and restore a highly-available database. Table 16–13 lists the `imqdbmgr` subcommands.

> **Note:** When using a cluster configuration file to simplify management of the properties for a cluster of brokers, make sure to run `imqdbmgr` with the `-D` option to specify the location of the cluster configuration file; for example:
>
> ```
> imqdbmgr
> -Dimq.cluster.url=location-of-cluster-configuration-file ...
> ```

Table 16–13   Database Manager Subcommands

Subcommand	Description
`create all`	Create new database and persistent data store schema
	Used on embedded database systems. The broker property `imq.persist.jdbc.`*vendorName*`.createdburl` must be specified.
`create tbl`	Create persistent data store schema for existing database
	Used on external database systems.
	For brokers belonging to an enhanced broker cluster (`imq.cluster.ha = true`), the schema created is for the cluster's shared data store, in accordance with the database vendor identified by the broker's `imq.persist.jdbc.dbVendor` property. If `imq.cluster.ha = false`, the schema is for the individual broker's standalone data store. Since the two types of data store can coexist in the same database, they are distinguished by appending a suffix to all table names:
	■ `C` *clusterID*: Shared data store
	■ `S` *brokerID*: Standalone data store
`delete tbl`	Delete Message Queue database tables from current data store
`delete oldtbl`	Delete Message Queue database tables from earlier-version data store
	Used after the data store has been automatically migrated to the current version of Message Queue.
`recreate tbl`	Re-create persistent store schema
	Deletes all existing Message Queue database tables from the current persistent store and then re-creates the schema.
`query`	Display information about the data store
`upgrade hastore`	Upgrade standalone data store to shared data store
`backup`	Back up JDBC-based data store to backup files
`restore`	Restore JDBC-based data store from backup files
`remove bkr`	Remove broker from shared data store
	The broker must not be running.
`remove jmsbridge`	Remove JMS bridge from the shared data store
	The broker hosting the JMS bridge must not be running.
`reset lck`	Reset data store lock
	Resets the lock so that the database can be used by other processes.
Subcommands for a Cluster's Shared Configuration Change Table	
`create sharecc_tbl`	Create the shared database table for the cluster configuration change record
`delete sharecc_tbl`	Delete the shared database table for the cluster configuration change record
`recreate sharecc_tbl`	Re-create the shared database table for the cluster configuration change record.
	Deletes existing shared database table and then re-creates it.
`backup sharecc_tbl-file` *filePath*	Back up the shared database table for the cluster configuration change record to a backup file
`restore sharecc_tbl-file` *filePath*	Restore the shared database table for the cluster configuration change record from a backup file. The table must already exist.

Table 16–14 lists the options to the `imqdbmgr` command.

Table 16–14   Database Manager Options

Option	Description
`-b instanceName`	Instance name of broker
`-Dproperty=value`	Set broker configuration property
	See Persistence Properties for information about persistence-related broker configuration properties.
	Caution: Be careful to check the spelling and formatting of properties set with this option. Incorrect values will be ignored without notification or warning.
`-u userName`	User name for authentication against the database
`-passfile filePath`	Location of password file
	See Password Files for more information.
`-n brokerID`	(Used with the `remove bkr` subcommand) Broker identifier of broker to be removed from shared data store
`-n bridgeName`	(Used with the `remove jmsbridge` subcommand) Bridge name of the JMS bridge to be removed from shared data store
`-dir dirPath`	Backup directory for backing up or restoring JDBC-based data store
`-v`	Display version information[1]
`-h`	Display usage help[1]

[1]   Any other options specified on the command line are ignored.

User Manager Utility

The User Manager utility (`imqusermgr`) is used for populating or editing a flat-file user repository. The utility must be run on the same host where the broker is installed; if a broker-specific user repository does not yet exist, you must first start up the corresponding broker instance in order to create it. You will also need the appropriate permissions to write to the repository: on the Solaris or Linux platforms, this means you must be either the root user or the user who originally created the broker instance.

Table 16–15 lists the subcommands available with the `imqusermgr` command. In all cases, the `-i` option specifies the instance name of the broker to whose user repository the command applies; if not specified, the default name `imqbroker` is assumed.

Table 16–15   User Manager Subcommands

Syntax	Description

`add [-i instanceName]`
`-u userName -p password`
`[-g  group]`
	Add user and password to repository

	The optional `-g` option specifies a group to which to assign this user:
	■ `admin`
	■ `user`
	■ `anonymous`

Table 16–15   (Cont.) User Manager Subcommands

Syntax	Description
decode -src *encodedPassfile* -target *unencodedPassfile*	Deobfuscates a passfile. ■ The -src option takes a string, represented by *encodedPassfile*, to specify the path and file name of the source file to be deobfuscated. ■ The -target option takes a string, represented by *unencodedPassfile*, to specify the path and file name of the resulting clear text file.
delete [-i *instanceName*] -u *userName* -target	Delete user from repository
encode -src *unencodedPassfile* -target *encodedPassfile*	Obfuscates a passfile. ■ The -src option takes a string, represented by *unencodedPassfile*, to specify the path and file name of the source file to be obfuscated. ■ The -target option takes a string, represented by *encodedPassfile*, to specify the path and file name of the resulting obfuscated file.
update [-i *instanceName*] -u *userName* -p *password* update [-i *instanceName*] -u *userName* -a *activeStatus* update [-i *instanceName*] -u *userName* -p *password* -a *activeStatus*	Set user's password or active status (or both) The -a option takes a boolean value specifying whether to make the user active (true) or inactive (false). An inactive status means that the user entry remains in the user repository, but the user will not be authenticated, even if using the correct password. **Default value:** true.
list [-i *instanceName*] [-u *userName*]	Display user information If no user name is specified, all users in the repository are listed.

In addition, the options listed in Table 16–16 can be applied to any subcommand of the imqusermgr command.

Table 16–16   General User Manager Options

Option	Description
-D*brokerProperty*=*value*	Specify a broker property value when starting imqusermgr.
-f	Perform action without user confirmation
-s	Silent mode (no output displayed)
-v	Display version information[1]
-h	Display usage help[1]

[1] Any other options specified on the command line are ignored.

Bridge Manager Utility

The Bridge Manager utility (imqbridgemgr) is used to manage the bridges configured for a broker, including the links within bridge types that support links. The basic syntax of imqbridgemgr is:

```
imqbridgemgr subcommand commandArgument [ options ]
imqbridgemgr -h | -help
```

```
imqbridgemgr -H | -Help
imqbridgemgr -v | -version
```

Table 16–17 lists the imqbridgemgr subcommands for general bridge management, Table 16–18 lists the imqbridgemgr subcommands for link management, which are applicable only to bridge types that support links, and Table 16–19 lists the imqbridgemgr options.

Table 16–17   Bridge Manager Subcommands for Bridge Management

Subcommand	Description
list bridge	Lists the bridges specified by the command options provided. For each bridge, the bridge name, type and state are displayed.
pause bridge	Pauses the bridges specified by the command options provided if the bridge type supports this subcommand.
	Attempting to pause a bridge that is stopped generates an error, and attempting to pause a bridge that is already paused has no effect.
resume bridge	Resumes the bridges specified by the command options provided if the bridge type supports this subcommand.
	Attempting to resume a bridge that is stopped generates an error, and attempting to resume a bridge that is already started has no effect.
start bridge	Starts the bridges specified by the command options provided.
	Attempting to start a bridge that is paused causes the bridge to resume, and attempting to start a bridge that is already started has no effect.
stop bridge	Stops the bridges specified by the command options provided.
	Attempting to stop a bridge that is paused causes the bridge to stop, and attempting to stop a bridge that is already stopped has no effect.

Table 16–18   Bridge Manager Subcommands for Link Management

Subcommand	Description
list link	Lists the links specified by the command options provided. For each link, the link name, state, source, target, and transaction status are displayed.
pause link	Pauses the link specified by the command options provided.
	Attempting to pause a link that is stopped, in the process of stopping, or has never been started generates an error. Attempting to pause a link that is already paused or in the process of pausing has no effect.
resume link	Resumes the link specified by the command options provided.
	Attempting to resume a link that is stopped, in the process of stopping, or has never been started generates an error. Attempting to resume a link that is already started or in the process of starting has no effect.
start link	Starts the link specified by the command options provided.
	Attempting to start a link that is paused causes the link to resume. Attempting to start a link that is in the process of pausing causes the link to complete pausing and then to resume. Attempting to start a link that is already started or in the process of starting has no effect.
stop link	Stops the link specified by the command options provided.
	Attempting to stop a link that has never been started generates an error. Attempting to stop a link that is in the process of starting causes the link to complete starting and then to stop. Attempting to stop a link that is paused causes the link to stop. Attempting to stop a link that is in the process of pausing causes the link to complete pausing and then to stop. Attempting to stop a link that is already stopped or in the process of stopping has no effect.

Table 16–19 lists the options to the `imqbridgemgr` command.

Table 16–19   Bridge Manager Options

Option	Description
-b *hostName*:*portNumber*	The broker housing the bridge.
	Literal IP addresses as host names: You can use a literal IPv4 or IPv6 address as a host name. If you use a literal IPv6 address, its format must conform to RFC2732 (http://www.ietf.org/rfc/rfc2732.txt), *Format for Literal IPv6 Addresses in URL's*.
	Default value: localhost:7676
-bn *bridgeName*	The name of the bridge.
-f	Perform the action without user confirmation
-javahome *path*	Location of an alternative Java runtime.
	Default behavior: Use the runtime installed with Message Queue.
-ln *linkName*	The name of the link.
-passfile *path*	Location of password file
-rtm *timeoutInterval*	Initial timeout interval, in seconds
	This is the initial length of time that the Command utility will wait for a reply from the broker before retrying a request. Each subsequent retry will use a timeout interval that is a multiple of this initial interval.
	Default value: 10
-rtr *numRetries*	Number of retries to attempt after a broker request times out
	Default value: 5
-s	Silent mode (no output displayed)
-secure	Use secure connection to broker with `ssladmin` connection service
-t *bridgeType*	The type of the bridge: `JMS` or `STOMP`
-u *userName*	User name for authentication

Service Administrator Utility

The Service Administrator utility (`imqsvcadmin`) installs a broker as a Windows service. Table 16–20 lists the available subcommands.

Table 16–20   Service Administrator Subcommands

Subcommand	Description
install	Install service
remove	Remove service
query	Display startup options
	Startup options can include whether the service is started manually or automatically, its location, the location of the Java runtime, and the values of arguments passed to the broker on startup (see Table 16–21).

Table 16–21 lists the options to the `imqsvcadmin` command.

Table 16–21   Service Administrator Options

Option	Description
-javahome *path*	Location of alternative Java runtime **Default behavior:** Use runtime installed on system or bundled with Message Queue.
-jrehome *path*	Location of alternative Java Runtime Environment (JRE)
-vmargs *arg1* [*arg2*]…	Additional arguments to pass to Java Virtual Machine (JVM) running broker service[1] **Example:** `imqsvcadmin install vmargs "-Xms16m -Xmx128m"`
-args *arg1* [*arg2*]…	Additional command line arguments to pass to broker service[1] **Example:** `imqsvcadmin install args "passfile d:\\imqpassfile"` See Broker Utility for information about broker command line arguments.
-h	Display usage help[2]

[1] These arguments can also be specified in the Start Parameters field under the General tab in the service's Properties window (reached by way of the Services tool in the Windows Administrative Tools control panel).

[2] Any other options specified on the command line are ignored.

Any information you specify using the -javahome, -vmargs, and -args options is stored in the Windows registry under the keys JREHome, JVMArgs, and ServiceArgs in the path

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\iMQ_Broker\Parameters`

Key Tool Utility

The Key Tool utility (imqkeytool) generates a self-signed certificate for the broker, which can be used for the ssljms, ssladmin, or cluster connection service. The syntax is

`imqkeytool -broker`

On UNIX systems, you might need to run the utility from the root user account.

17

Broker Properties Reference

This chapter provides reference information about configuration properties for a message broker. It consists of the following sections:

- Connection Properties
- Routing and Delivery Properties
- Persistence Properties
- Security Properties
- Monitoring Properties
- Cluster Configuration Properties
- Bridge Properties
- JMX Properties
- Alphabetical List of Broker Properties

Connection Properties

Table 17–1 lists the broker properties related to connection services.

Table 17–1   Broker Connection Properties

Property	Type	Default Value	Description
`imq.brokerid`	String	None	Broker identifier
			For brokers using a shared JDBC-based data store, this string is appended to the names of all database tables to identify each table with a particular broker.
			Must be a unique alphanumeric string of no more than n - 18 characters, where n is the maximum table name length allowed by the database.
			This property is unnecessary for an embedded database or a standalone database which stores data for only one broker instance.
			Note: For enhanced broker clusters (`imq.cluster.ha = true`), database table names also use the `imq.cluster.clusterid` property (see Table 17–14).
`imq.service.activelist`[1]	String	`jms,admin`	List of connection services to be activated at broker startup, separated by commas
			See Table 6–1 under Configuring Connection Services for the names of the available connection services.
`imq.hostname`	String	All available IP addresses	Default host name or IP address for all connection services
`imq.portmapper.hostname`	String	None	Host name or IP address of Port Mapper
			If specified, overrides `imq.hostname`. This might be necessary, for instance, if the broker's host computer has more than one network interface card installed.
`imq.portmapper.port`[2]	Integer	`7676`	Port number of Port Mapper
			Note: If multiple broker instances are running on the same host, each must be assigned a unique Port Mapper port.
`imq.`*serviceName*`.`*protocolType*`.hostname` [3]	String	None	Host name or IP address for connection service
			If specified, overrides `imq.hostname` for the designated connection service. This might be necessary, for instance, if the broker's host computer has more than one network interface card installed.
`imq.`*serviceName*`.`*protocolType*`.port`[3]	Integer	`0`	Port number for connection service
			A value of `0` specifies that the port number should be allocated dynamically by the Port Mapper. You might need to set a different value, for instance, to specify a static port number for connecting to the broker through a firewall.
`imq.portmapper.backlog`	Integer	`50`	Maximum number of pending Port Mapper requests in operating system backlog

Table 17–1   (Cont.) Broker Connection Properties

Property	Type	Default Value	Description
imq.*serviceName*.threadpool_model[4]	String	dedicated	Threading model for thread pool management: ■ dedicated: Two dedicated threads per connection, one for incoming and one for outgoing messages ■ shared: Connections processed by shared thread when sending or receiving messages The dedicated model limits the number of connections that can be supported, but provides higher performance; the shared model increases the number of possible connections, but at the cost of lower performance because of the additional overhead needed for thread management.
imq.*serviceName*.min_threads	Integer	jms: 10ssljms: 10httpjms: 10 httpsjms: 10 admin: 4ssladmin: 4	Minimum number of threads maintained in connection service's thread pool When the number of available threads exceeds this threshold, threads will be shut down as they become free until the minimum is reached. The default value varies by connection service, as shown.
imq.*serviceName*.max_threads	Integer	jms: 1000ssljms: 500httpjms: 500httpsjms: 500admin: 10ssladmin: 10	Number of threads beyond which no new threads are added to the thread pool for use by the named connection service Must be greater than 0 and greater than the value of imq.*serviceName*.min_threads. The default value varies by connection service, as shown.
imq.shared.connectionMonitor_limit[5]	Integer	Solaris: 512Linux: 512Windows: 64	Maximum number of connections monitored by a distributor thread The system allocates enough distributor threads to monitor all connections. The smaller the value of this property, the faster threads can be assigned to active connections. A value of -1 denotes an unlimited number of connections per thread. The default value varies by operating-system platform, as shown.
imq.ping.interval	Integer	120	Interval, in seconds, at which to test connection between client and broker A value of 0 or -1 disables periodic testing of the connection.

[1] Must have the same value for all brokers in an enhanced cluster.

[2] Can be used with imqcmd update bkr command

[3] jms, ssljms, admin, and ssladmin services only; see HTTP/HTTPS Support for information on configuring the httpjms and httpsjms services

[4] jms and admin services only

[5] Shared threading model only

Routing and Delivery Properties

This section includes the following tables:

- Table 17–2 lists the broker properties related to routing and delivery services
- Table 17–3 lists the broker properties that configure the automatic creation of destinations
- Table 17–4 lists the broker properties that configure the administrative creation of destinations

Table 17–2   Broker Routing and Delivery Properties

Property	Type	Default Value	Description
imq.system.max_count[1]	Integer	-1	Maximum number of messages held by broker
			A value of -1 denotes an unlimited message count.
imq.system.max_size[1]	String	-1	Maximum total size of messages held by broker
			The value may be expressed in bytes, kilobytes, or megabytes, using the following suffixes:
			- b: Bytes
			- k: Kilobytes (1024 bytes)
			- m: Megabytes (1024 × 1024 = 1,048,576 bytes)
			An unsuffixed value is expressed in bytes; a value of -1 denotes an unlimited message capacity.
			Examples:
			- 1600: 1600 bytes
			- 1600b: 1600 bytes
			- 16k: 16 kilobytes (= 16,384 bytes)
			- 16m: 16 megabytes (= 16,777,216 bytes)
			- -1: No limit
imq.message.max_size[1]	String	70m	Maximum size of a single message body
			The syntax is the same as for imq.system.max_size (see above).
imq.message.expiration.interval	Integer	60	Interval, in seconds, at which expired messages are removed.
imq.*resourceState*.threshold	Integer	green: 0yellow: 80orange: 90red: 98	Percent utilization at which memory resource state is triggered (where *resourceState* is green, yellow, orange, or red)
imq.*resourceState*.count	Integer	green: 5000yellow: 500orange: 50red: 0	Maximum number of incoming messages allowed in a batch before checking whether memory resource state threshold has been reached (where *resourceState* is green, yellow, orange , or red)
			This limit throttles back message producers as system memory becomes increasingly scarce.
imq.destination.DMQ.truncateBody[1]	Boolean	false	Remove message body before storing in dead message queue?
			If true, only the message header and property data will be saved.
imq.transaction.autorollback	Boolean	false	Automatically roll back distributed transactions left in prepared state at broker startup?
			If false, transactions must be manually committed or rolled back using the Command utility (imqcmd).

Table 17–2   (Cont.) Broker Routing and Delivery Properties

Property	Type	Default Value	Description
imq.transaction.producer.maxNumMsgs	Integer	1000	The maximum number of messages that a producer can process in a single transaction. It is recommended that the value be less than 5000 to prevent the exhausting of resources.
imq.transaction.consumer.maxNumMsgs	Integer	100	The maximum number of messages that a consumer can process in a single transaction. It is recommended that the value be less than 1000 to prevent the exhausting of resources.
imq.transaction.message.maxConsecutiveRollbacks	Integer	0	Specifies the number of consecutive number of rollbacks that may occur before a broker automatically puts a message to DMQ, If there is more than one message in the transaction, only the last message in the transaction is placed in DMQ. The transaction is then rolled back and returns the error status. If the value of this property is <= 0, this feature is disabled.
imq.txn.reapInterval	Integer	900	The interval, in seconds, the reaper thread periodically runs to reap/cleanup committed transactions.
imq.txt.reapLimit	Integer	500 or 0 for an embedded broker	The maximum number of last committed transactions to retain when reaping/cleaning up committed transactions.

[1]   Can be used with imqcmd update bkr command

This table lists properties that configure the automatic creation of destinations.

Table 17–3   Broker Properties for Auto-Created Destinations

Property	Type	Default Value	Description
imq.autocreate.queue[1,2]	Boolean	true	Allow auto-creation of queue destinations?
imq.autocreate.topic[3]	Boolean	true	Allow auto-creation of topic destinations?
imq.autocreate.reaptime	Integer	120 seconds	The delay, in seconds. before which auto-created destinations are removed from the system when they no longer have consumers nor contain messages, . A smaller value means that memory reclamation takes place more often.

Table 17–3   (Cont.) Broker Properties for Auto-Created Destinations

Property	Type	Default Value	Description
`imq.autocreate.destination.maxNumMsgs`	Integer	100000	Maximum number of unconsumed messages
			A value of `-1` denotes an unlimited number of messages.
			Note: When flow control is in effect (`imq.autocreate.destination.limitBehavior` = `FLOW_CONTROL`), it is possible for the specified message limit to be exceeded because the broker cannot react quickly enough to stop the flow of incoming messages. In such cases, the value specified for `imq.autocreate.destination.maxNumMsgs` serves as merely a hint for the broker rather than a strictly enforced limit. However, if the number of unconsumed messages would exceed `imq.system.max_count`, the broker generates a `ResourceAllocationException` indicating that the destination is full and rejecting new messages.
`imq.autocreate.destination.maxBytesPerMsg`	String	10k	Maximum size, in bytes, of any single message
			The value may be expressed in bytes, kilobytes, or megabytes, using the following suffixes:
			■ `b`: Bytes
			■ `k`: Kilobytes (1024 bytes)
			■ `m`: Megabytes (1024 × 1024 = 1,048,576 bytes)
			An unsuffixed value is expressed in bytes; a value of `-1` denotes an unlimited message size.
			Examples:
			■ `1600`: 1600 bytes
			■ `1600b`: 1600 bytes
			■ `16k`: 16 kilobytes (= 16,384 bytes)
			■ `16m`: 16 megabytes (= 16,777,216 bytes)
			■ `-1`: No limit
`imq.autocreate.destination.maxTotalMsgBytes`	String	10m	Maximum total memory, in bytes, for unconsumed messages
			The syntax is the same as for `imq.autocreate.destination.maxBytesPerMsg` (see above).

Table 17–3   (Cont.) Broker Properties for Auto-Created Destinations

Property	Type	Default Value	Description
`imq.autocreate.destination.limitBehavior`	String	`REJECT_ NEWEST`	Broker behavior when memory-limit threshold reached: ■ `FLOW_CONTROL`: Slow down producers ■ `REMOVE_OLDEST`: Throw out oldest messages ■ `REMOVE_LOW_PRIORITY`: Throw out lowest-priority messages according to age; no notification to producing client ■ `REJECT_NEWEST`: Reject newest messages; notify producing client with an exception only if message is persistent When `FLOW_CONTROL` is specified, it is still possible for the number of messages to exceed `imq.system.max_count`. In this situation, the broker generates a `ResourceAllocationException` indicating that the destination is full and rejecting new messages.
			If the value is `REMOVE_OLDEST` or `REMOVE_LOW_PRIORITY` and the `imq.autocreate.destination.useDMQ` property is `true`, excess messages are moved to the dead message queue.
`imq.autocreate.destination.maxNumProducers`	Integer	100	Maximum number of message producers for destination When this limit is reached, no new producers can be created. A value of `-1` denotes an unlimited number of producers.
`imq.autocreate.queue.maxNumActiveConsumers`[2]	Integer	-1	Maximum number of active message consumers in load-balanced delivery from queue destination A value of `-1` denotes an unlimited number of consumers.
`imq.autocreate.queue.maxNumBackupConsumers`[2]	Integer	0	Maximum number of backup message consumers in load-balanced delivery from queue destination A value of `-1` denotes an unlimited number of consumers.

Table 17–3   (Cont.) Broker Properties for Auto-Created Destinations

Property	Type	Default Value	Description
`imq.autocreate.queue.consumerFlowLimit`[2]	Integer	1000	Maximum number of messages delivered to queue consumer in a single batch
			In load-balanced queue delivery, this is the initial number of queued messages routed to active consumers before load balancing begins. A destination consumer can override this limit by specifying a lower value on a connection.
			A value of `0` or `-1` denotes an unlimited number of messages.
`imq.autocreate.topic.consumerFlowLimit`[3]	Integer	1000	Maximum number of messages delivered to topic consumer in a single batch
			A value of `0` or `-1` denotes an unlimited number of messages.
			Not used when the JMS resource adapter, jmsra, is used to consume messages in a GlassFish Server cluster.
`imq.autocreate.topic.sharedConsumerFlowLimit`[3]	Integer	5	Maximum number of messages prefeteched to a topic shared subscriber or a topic consumer in a GlassFish server cluster.
			A value of `0` or `-1` denotes an unlimited number of messages.
`imq.autocreate.destination.isLocalOnly`	Boolean	false	Local delivery only?
			This property applies only to destinations in broker clusters, and cannot be changed once the destination has been created. If `true`, the destination is not replicated on other brokers and is limited to delivering messages only to local consumers (those connected to the broker on which the destination is created).
`imq.autocreate.queue.localDeliveryPreferred`[2]	Boolean	false	Local delivery preferred?
			This property applies only to load-balanced queue delivery in broker clusters. If `true`, messages will be delivered to remote consumers only if there are no consumers on the local broker; the destination must not be restricted to local-only delivery (`imq.autocreate.destination.isLocalOnly` must be `false`).
`imq.autocreate.destination.useDMQ`	Boolean	true	Send dead messages to dead message queue?
			If `false`, dead messages will simply be discarded.

Table 17–3   (Cont.) Broker Properties for Auto-Created Destinations

Property	Type	Default Value	Description
`validateXMLSchemaEnabled`	Boolean	`false`	XML schema validation is enabled?
			If set to `false` or not set, then XML schema validation is not enabled for the destination.
`XMLSchemaURIList`	String	null	Space separated list of XML schema document (XSD) URI strings
			The URIs point to the location of one or more XSDs to use for XML schema validation, if enabled.
			Use double quotes around this value if multiple URIs are specified.
			Example:
			`"http://foo/flap.xsd http://test.com/test.xsd"`
			If this property is not set or null and XML validation is enabled, XML validation is performed using a DTD specified in the XML document.
`reloadXMLSchemaOnFailure`	Boolean	`false`	Reload XML schema on failure enabled?
			If set to false or not set, then the schema is not reloaded if validation fails.

[1]   Can be used with `imqcmd update bkr` command
[2]   Queue destinations only
[3]   Topic destinations only

This table lists properties that apply to all administratively created destinations. They cannot be configured on individual administratively created destinations.

Table 17–4   Broker Properties for Admin-Created Destinations

Property	Type	Default Value	Description
`imq.admincreate.topic.sharedConsumerFlowLimit`	Integer	5	Maximum number of messages prefetched to a topic shared subscriber or to a topic consumer in a GlassFish Server cluster.
			A value of `0` or `-1` denotes an unlimited number of messages.

Persistence Properties

Message Queue supports both file-based and JDBC-based persistence modules. The broker property `imq.persist.store` (Table 17–5) specifies which module to use. The following sections describe the broker configuration properties for the two modules.

Table 17–5   Global Broker Persistence Property

Property	Type	Default Value	Description
`imq.persist.store`	String	`file`	Module used for persistent data storage:
			■ `file`: File-based persistence
			■ `jdbc`: JDBC-based persistence
			Must be set to `jdbc` for enhanced broker clusters (`imq.cluster.ha` = `true`).

File-Based Persistence Properties

Table 17–6 lists the broker properties related to file-based persistence.

Table 17–6   Broker Properties for File-Based Persistence

Property	Type	Default Value	Description
`imq.persist.file.message.max_record_size`	String	1m	Maximum-size message to add to message storage file
			Any message exceeding this size will be stored in a separate file of its own.
			The value may be expressed in bytes, kilobytes, or megabytes, using the following suffixes:
			■ `b`: Bytes
			■ `k`: Kilobytes (1024 bytes)
			■ `m`: Megabytes (1024 × 1024 = 1,048,576 bytes)
			An unsuffixed value is expressed in bytes.
			Examples:
			■ `1600`: 1600 bytes
			■ `1600b`: 1600 bytes
			■ `16k`: 16 kilobytes (= 16,384 bytes)
			■ `16m`: 16 megabytes (= 16,777,216 bytes)
`imq.persist.file.destination.message.filepool.limit`	Integer	100	Maximum number of free files available for reuse in destination file pool
			Free files in excess of this limit will be deleted. The broker will create and delete additional files in excess of the limit as needed.
			The higher the limit, the faster the broker can process persistent data.
`imq.persist.file.message.filepool.cleanratio`	Integer	0	Percentage of files in free file pools to be maintained in a clean (empty) state
			The higher this value, the less disk space is required for the file pool, but the more overhead is needed to clean files during operation.

Table 17–6   (Cont.) Broker Properties for File-Based Persistence

Property	Type	Default Value	Description
`imq.persist.file.message.cleanup`	Boolean	`false`	Clean up files in free file pools on shutdown?
			Setting this property to `true` saves disk space for the file store, but slows broker shutdown.
`imq.persist.file.sync.enabled`	Boolean	`false`	Synchronize in-memory state with physical storage device?
			Setting this property to `true` eliminates data loss due to system crashes, but at a cost in performance.
			Note: If running Oracle Solaris Cluster and its Data Service for Message Queue, set this property to `true` for brokers on all cluster nodes.
`imq.persist.file.transaction.memorymappedfile.enabled`	Boolean	`true`	Use memory-mapped file to store transaction data?
			Setting this property to `true` improves performance at the cost of increased memory usage. Set to `false` for file systems that do not support memory-mapped files.

File-Based Persistence Properties for Transaction Logging

Table 17–7 lists the file-based persistence properties for the transaction logging mechanism.

Table 17–7   Broker Properties for File-Based Persistence Using the Transaction Logging Mechanism

Property	Type	Default Value	Description
`imq.persist.file.newTxnLog.enabled`	Boolean	`false`	Enables the transaction logging mechanism. For information about this mechanism, see Optimizing File-Based Transaction Persistence.

Table 17–7   (Cont.) Broker Properties for File-Based Persistence Using the Transaction Logging

Property	Type	Default Value	Description
`imq.persist.file.txnLog.groupCommit`	Boolean	`false`	This property is applicable only if `imq.persist.file.newTxnLog.enabled` is `true`.
			Can improve performance if `imq.persist.file.sync.enabled` is `true` and the number of concurrent transactions being processed is high:
			■ If `true`, write operations to the transaction log are not handled by individual connection threads; instead, writes from connection threads are added to a transaction queue. The connection threads then wait until they are notified that the transactions have been logged. A separate thread periodically drains the transaction queue and writes it to the transaction log. When possible, this thread groups together multiple active transactions and writes them to the transaction log in a single operation. After the write completes, waiting client threads are notified.
			■ If `false`, write operations to the transaction log are handled by individual connection threads. Only one thread at a time is able to write to the log.
`imq.persist.file.txnLog.logNonTransactedMsgSend`	Boolean	`false`	This property is applicable only if `imq.persist.file.newTxnLog.enabled` is `true`.
			Overrides the behavior for persisting non-transacted messages (as defined by the `imq.persist.file.sync.enabled` property):
			■ If `true`, non-transacted messages are written to the transaction log before they are written to the persistent store.
			■ If `false`, non-transacted messages are written directly to the persistent store.

Table 17–7 (Cont.) Broker Properties for File-Based Persistence Using the Transaction Logging

Property	Type	Default Value	Description
`imq.persist.file.txnLog.logNonTransactedMsgAck`	Boolean	false	This property is applicable only if `imq.persist.file.newTxnLog.enabled` is `true`.
			Overrides the behavior for persisting non-transacted message acknowledgements (as defined by the `imq.persist.file.sync.enabled` property):
			■ If `true`, acknowledgements of non-transacted messages are written to the transaction log before they are written to the persistent store.
			■ If `false`, acknowledgements of non-transacted messages are written directly to the persistent store.

JDBC-Based Persistence Properties

Table 17–8 lists the broker properties related to JDBC-based persistence. The first of these properties, `imq.persist.jdbc.dbVendor`, identifies the database vendor being used for the broker's persistent data store; all of the remaining properties are qualified by this vendor name.

Table 17–8   Broker Properties for JDBC-Based Persistence

Property	Type	Default Value	Description
`imq.persist.jdbc.dbVendor`	String	None	Name of database vendor for persistent data store:
			■ `derby`: Java DB (Oracle Corporation)
			■ `oracle`: Oracle (Oracle Corporation)
			■ `mysql`: MySQL (Oracle Corporation)
			■ `postgresql`: postgreSQL
`imq.persist.jdbc.connection.retry.delay`	Integer	5000	The amount of time, in milliseconds, the broker waits before it attempts to retry connecting to a database server for a failed connection.
`imq.persist.jdbc.connection.retry.max`	Integer	60	The maximum number of times the broker will retry connecting to a database server after a failed connection.
`imq.persist.jdbc.connection.reaptime`	Integer	300	The interval in seconds between attempts to close unnecessary database connections.
`imq.persist.jdbc.max_connections`	Integer	5	The maximum number of connections that should be opened to the database. The Message Queue database connection pool manager uses this value as a guide when creating new connections to the database.

Table 17–8   (Cont.) Broker Properties for JDBC-Based Persistence

Property	Type	Default Value	Description
`imq.persist.jdbc.min_connections`	Integer	5	The number of connections that are opened to the database when the Message Queue database connection pool is initialized, and the minimum number of connections that are to be kept open when unnecessary connections are closed.
`imq.persist.jdbc.connection.timeoutIdle`	Boolean	true	Should the Message Queue database connection pool manager considers the age of a connection in the pool? When `true`, the pool manager operates as follows: ■ When closing unnecessary database connections at each `imq.persist.jdbc.connection.reaptime` interval, the pool manager destroys any connections in the pool that have been idle for a period longer than `imq.persist.jdbc.connection.reaptime` seconds, and attempts to replace them with new connections. ■ When getting a connection from the pool, the pool manager destroys the connection if it has been idle for a period longer than `imq.persist.jdbc.connection.reaptime` seconds and replaces it with a new connection.
`imq.persist.jdbc.connection.validateOnGet`	Boolean	true for brokers in enhanced clusters; otherwise, false	Should the Message Queue database connection pool manager perform extra validation when it gets a connection from the pool? Normally, the pool manager validates a connection it gets from the pool by checking whether the connection is closed and, if possible, by checking whether a connection error has occurred on the connection. When this property is set to `true`, the pool manager performs these additional validation checks: ■ For JDBC 4 connections, check whether the connection is valid. ■ Make the validation query specified by `imq.persist.jdbc.connection.validationQuery` on the connection. Regardless of the value of this property, Message Queue relies on the underlying JDBC driver to return a good connection when it needs to create a new connection through the driver. Therefore, you should set `imq.persist.jdbc.`*`vendorName`*`.driver` to a vendor JDBC driver that performs connection retries when creating a new connection to the database server; for example, a ConnectionPoolDataSource.

Table 17–8   (Cont.) Broker Properties for JDBC-Based Persistence

Property	Type	Default Value	Description
`imq.persist.jdbc.connection.validationQuery`	String	per vendor	The validation query for the Message Queue database connection pool manager to use when `imq.persist.jdbc.connection.validateOnGet` is set to `true`.
`imq.persist.jdbc.connection.limit`	Integer	5	The maximum number of connections that can be opened to the database. This property is deprecated and may become unsupported in a future release of Message Queue.
`imq.persist.jdbc.`*vendorName*`.connection.retry.regex.`*n*	String	(?s).*	There can be 0 or many of these properties specified, where *n* represents the nth such property. Each of these properties specifies a regular expression that can be used to match a `SQLException` message in order to identify retriable database communication failures. These regular expressions are parsed in consecutive order starting with 1 and stops when a match is found.
`imq.persist.jdbc.`*vendorName*`.driver`	String	per vendor	Java class name of JDBC driver, if needed, for connecting to database from vendor *vendorName*
`imq.persist.jdbc.`*vendorName*`.opendburl`	String	None	URL for connecting to existing database from vendor *vendorName* Applicable when driver is used to connect to database.
`imq.persist.jdbc.`*vendorName*`.createdburl`[1]	String	None	URL for creating new database from vendor *vendorName* Applies for embedded database, such as Java DB.
`imq.persist.jdbc.`*vendorName*`.closedburl`[1]	String	None	URL for closing connection to database from vendor *vendorName* Applies for some embedded databases, such as Java DB.
`imq.persist.jdbc.`*vendorName*`.user`[1]	String	None	User name, if required, for connecting to database from vendor *vendorName* For security reasons, the value can instead be specified using command line options `imqbrokerd dbuser` and `imqdbmgr u`.
`imq.persist.jdbc.`*vendorName*`.needpassword`[1]	Boolean	false	Does database from vendor *vendorName* require a password for broker access? If `true`, the `imqbrokerd` and `imqdbmgr` commands will prompt for a password, unless you use the `-passfile` option to specify a password file containing it.

Table 17–8 (Cont.) Broker Properties for JDBC-Based Persistence

Property	Type	Default Value	Description
imq.persist.jdbc.*vendorName*.password[1,2]	String	None	Password, if required, for connecting to database from vendor *vendorName*
imq.persist.jdbc.*vendorName*.property.*propName*[1]	String	None	Vendor-specific property *propName* for database from vendor *vendorName*
imq.persist.jdbc.*vendorName*.tableoption[1]	String	None	Vendor-specific options passed to the database when creating the table schema.

[1] Optional
[2] Should be used only in password files

Security Properties

Table 17–9 lists broker properties related to security services: authentication, authorization, and encryption. Table 17–10 lists broker properties related specifically to file-based authentication, Table 17–11 lists broker properties related specifically to LDAP-based authentication, and Table 17–12 lists broker properties related specifically to JAAS-based authentication.

Table 17–9   Broker Security Properties

Property	Type	Default Value	Description
imq.authentication.basic.user_repository	String	file	Type of user authentication: ■ file: File-based ■ ldap: Lightweight Directory Access Protocol ■ jaas: Java Authentication and Authorization Service
imq.authentication.type	String	digest	Password encoding method: ■ digest: MD5 (for file-based authentication) ■ basic: Base-64 (for LDAP or JAAS authentication)
imq.*serviceName*.authentication.type	String	None	Password encoding method for connection service *serviceName*: ■ digest: MD5 (for file-based authentication) ■ basic: Base-64 (for LDAP or JAAS authentication) If specified, overrides imq.authentication.type for the designated connection service.

Table 17–9 (Cont.) Broker Security Properties

Property	Type	Default Value	Description
`imq.authentication.client.response.timeout`	Integer	`180`	Interval, in seconds, to wait for client response to authentication requests
`imq.accesscontrol.enabled`	Boolean	`true`	Use access control?
			If `true`, the system will check the access control file to verify that an authenticated user is authorized to use a connection service or to perform specific operations with respect to specific destinations.
`imq.accesscontrol.type`	String	`file`	Specifies the access control type
`imq.`*serviceName*`.accesscontrol.enabled`	Boolean	None	Use access control for connection service?
			If specified, overrides `imq.accesscontrol.enabled` for the designated connection service.
			If `true`, the system will check the access control file to verify that an authenticated user is authorized to use the designated connection service or to perform specific operations with respect to specific destinations.
`imq.accesscontrol.file.dirpath`	String	`IMQ_ VARHOME/instances/ `*instanceName*`/etc`	Path to the access control directory
`imq.accesscontrol.file.filename`	String	`accesscontrol.propert ies`	Name of access control file
			The file name specifies a path relative to `imq.accesscontrol.file.di rpath`.
`imq.`*serviceName*`.accesscontrol.file.filename`	String	None	Name of access control file for connection service
			If specified, overrides `imq.accesscontrol.file.fi lename` for the designated connection service.
			The file name specifies a path relative to `imq.accesscontrol.file.di rpath`.

Table 17–9   (Cont.) Broker Security Properties

Property	Type	Default Value	Description
`imq.accesscontrol.file.url`	String	Not set	The location, as a URL, of the access control file.
			If the URL uses LDAP protocol (`ldap://`), the access control file must be returned as a single string that uses dollar sign (`$`) as the separator between the lines of the access control file.
`imq.`*serviceName*`.accesscontrol.file.url`	String	None	The location, as a URL, of the access control file for the connection service.
			If specified, overrides `imq.accesscontrol.file.url` for the designated connection service.
			If the URL uses LDAP protocol (`ldap://`), the access control file must be returned as a single string that uses dollar sign (`$`) as the separator between the lines of the access control file.
`imq.keystore.file.dirpath`	String	`IMW_HOME/etc`	Path to directory containing key store file
`imq.keystore.file.name`	String	`keystore`	Name of key store file
`imq.keystore.password`[1]	String	None	Password for key store file
`imq.passfile.enabled`	Boolean	`false`	Obtain passwords from password file?
`imq.passfile.dirpath`	String	`IMQ_HOME/etc`	Path to directory containing password file
`imq.passfile.name`	String	`passfile`	Name of password file
`imq.imqcmd.password`[1]	String	None	Password for administrative user
			The Command utility (`imqcmd`) uses this password to authenticate the user before executing a command.
`imq.audit.enabled`	Boolean	`false`	Is audit logging to broker log file enabled?
`imq.audit.bsm.disabled`	Boolean	`true`	Is audit logging to the Solaris BSM audit log disabled?

[1]   To be used only in password files

Table 17–10 lists broker properties related to user authentication when using a flat-file user repository.

Table 17–10   Broker Security Properties for Flat-File Authentication

Property	Type	Default Value	Description
`imq.user_repository.file.dirpath`	String	`IMQ_` `VARHOME/instances/` *`instanceName`*`/etc/`	Path to the directory containing the flat-file user repository
`imq.user_repository.file.filename`	String	`passwd`	Name of the flat-file user repository file in the directory specified by `imq.user_repository.file.dirpath`

Table 17–11 lists broker properties related to LDAP-based user authentication.

Table 17–11   Broker Security Properties for LDAP Authentication

Property	Type	Default Value	Description
`imq.user_repository.ldap.server`	String	None	Host name and port number for LDAP server
			The value is of the form
			■ *hostName*:*port*
			where *hostName* is the fully qualified DNS name of the host running the LDAP server and *port* is the port number used by the server.
			To specify a list of failover servers, use the following syntax:
			■ *host1*:*port1*
			■ `ldap://`*host2*: *port2*
			■ `ldap://`*host3* :*port3*
			■ …
			Entries in the list are separated by spaces. Note that each failover server address is prefixed with `ldap://`. Use this format even if you use SSL and have set the property `imq.user_repository.ldap.ssl.enabled` to `true`. You need not specify `ldaps` in the address.
`imq.user_repository.ldap.principal`	String	None	Distinguished name for binding to LDAP user repository
			Not needed if the LDAP server allows anonymous searches.
`imq.user_repository.ldap.password`[1]	String	None	Password for binding to LDAP user repository
			Not needed if the LDAP server allows anonymous searches.
`imq.user_repository.ldap.`*propertyName*			

Table 17–11 (Cont.) Broker Security Properties for LDAP Authentication

Property	Type	Default Value	Description
`imq.user_repository.ldap.base`	String	None	Directory base for LDAP user entries
`imq.user_repository.ldap.uidattr`	String	None	Provider-specific attribute identifier for LDAP user name
`imq.user_repository.ldap.usrformat`	String	None	When set to a value of `dn`, specifies that DN username format is used for authentication (for example: `uid=mquser, ou=People, dc=red, dc=sun, dc=com`).
			Also, the broker extracts the value of the `imq.user.repository.lpdap.uidatr` attribute from the DN username, and uses this value as the user name in access control operations.
			If not set, then normal username format is used.
`imq.user_repository.ldap.usrfilter`[2]	String	None	JNDI filter for LDAP user searches
`imq.user_repository.ldap.grpsearch`	Boolean	`false`	Enable LDAP group searches?
			Note: Message Queue does not support nested groups.
`imq.user_repository.ldap.grpbase`	String	None	Directory base for LDAP group entries
`imq.user_repository.ldap.gidattr`	String	None	Provider-specific attribute identifier for LDAP group name
`imq.user_repository.ldap.memattr`	String	None	Provider-specific attribute identifier for user names in LDAP group
`imq.user_repository.ldap.grpfilter`[2]	String	None	JNDI filter for LDAP group searches
`imq.user_repository.ldap.timeout`	Integer	`280`	Time limit for LDAP searches, in seconds
`imq.user_repository.ldap.ssl.enabled`	Boolean	`false`	Use SSL when communicating with LDAP server?
`imq.user_repository.ldap.ssl.socketfactory`	String	`com.sun.messaging.jmq.jmsserver.auth.ldap.TrustSSLSocketFactory`	The fully qualified class name of the socket factory to use to make SSL connections to the LDAP server.
			When this property is not set and `imq.user_repository.ldap.ssl.enabled` is set to `true`, the default socket factory designated by the LDAP naming service is used.

[1] Should be used only in password files
[2] Optional

Table 17–12 lists broker properties related to JAAS-based user authentication.

Table 17–12   Broker Security Properties for JAAS Authentication

Property	Type	Default Value	Description
`imq.user_repository.jaas.name`	String	None	Set to the name of the desired entry (in the JAAS configuration file) that references the login modules you want to use as the authentication service.
`imq.user_ repository.jaas.userPrincipalClass`	String	None	This property, used by Message Queue access control, specifies the `java.security.Principal` implementation class in the login module(s) that the broker uses to extract the Principal name to represent the user entity in the Message Queue access control file. If, it is not specified, the user name passed from the Message Queue client when a connection was requested is used instead.
`imq.user_ repository.jaas.groupPrincipalClass`	String	None	This property, used by Message Queue access control, specifies the `java.security.Principal` implementation class in the login module(s) that the broker uses to extract the Principal name to represent the group entity in the Message Queue access control file. If, it is not specified, the user name passed from the Message Queue client when a connection was requested is used instead.

Monitoring Properties

Table 17–13 lists the broker properties related to monitoring services.

Table 17–13   Broker Monitoring Properties

Property	Type	Default Value	Description
`imq.log.level`[1]	String	`INFO`	Logging level
			Specifies the categories of logging information that can be written to an output channel. Possible values, from high to low:
			▪ `ERROR`
			▪ `WARNING`
			▪ `INFO`
			Each level includes those above it (for example, `WARNING` includes `ERROR`).
`imq.destination.logDeadMsgs`[1]	Boolean	`false`	Log information about dead messages?
			If `true`, the following events will be logged:
			▪ A destination is full, having reached its maximum size or message count.
			▪ The broker discards a message for a reason other than an administrative command or delivery acknowledgment.
			▪ The broker moves a message to the dead message queue.
`imq.log.console.stream`	String	`ERR`	Destination for console output:
			▪ `OUT`: `stdout`
			▪ `ERR`: `stderr`
`imq.log.console.output`	String	`ERROR|WARNING`	Categories of logging information to write to console:
			▪ `NONE`
			▪ `ERROR`
			▪ `WARNING`
			▪ `INFO`
			▪ `ALL`
			The `ERROR`, `WARNING`, and `INFO` categories do *not* include those above them, so each must be specified explicitly if desired. Any combination of categories can be specified, separated by vertical bars (`|`).
`imq.log.file.dirpath`	String	`IMQ_VARHOME/instances/` *instanceName*`/log`	Path to directory containing log file
`imq.log.file.filename`	String	`log.txt`	Name of log file

Table 17–13   (Cont.) Broker Monitoring Properties

Property	Type	Default Value	Description
imq.log.file.output	String	ALL	Categories of logging information to write to log file: ■ NONE ■ ERROR ■ WARNING ■ INFO ■ ALL The ERROR, WARNING, and INFO categories do *not* include those above them, so each must be specified explicitly if desired. Any combination of categories can be specified, separated by vertical bars (|).
imq.log.file.rolloverbytes[1]	Integer	-1	File length, in bytes, at which output rolls over to a new log file A value of -1 denotes an unlimited number of bytes (no rollover based on file length).
imq.log.file.rolloversecs[1]	Integer	604800 (one week)	Age of file, in seconds, at which output rolls over to a new log file A value of -1 denotes an unlimited number of seconds (no rollover based on file age).
imq.log.syslog.output[2]	String	ERROR	Categories of logging information to write to syslogd(1M): ■ NONE ■ ERROR ■ WARNING ■ INFO ■ ALL The ERROR, WARNING, and INFO categories do *not* include those above them, so each must be specified explicitly if desired. Any combination of categories can be specified, separated by vertical bars (|).

Table 17–13   (Cont.) Broker Monitoring Properties

Property	Type	Default Value	Description
`imq.log.syslog.facility`[2]	String	`LOG_DAEMON`	`syslog` facility for logging messages
			Possible values mirror those listed on the `syslog(3C)` man page. Appropriate values for use with Message Queue include:
			■ `LOG_USER`
			■ `LOG_DAEMON`
			■ `LOG_LOCAL0`
			■ `LOG_LOCAL1`
			■ `LOG_LOCAL2`
			■ `LOG_LOCAL3`
			■ `LOG_LOCAL4`
			■ `LOG_LOCAL5`
			■ `LOG_LOCAL6`
			■ `LOG_LOCAL7`
`imq.log.syslog.identity`[2]	String	`imqbrokerd_ ${imq.`*instanceName*`}`	Identity string to be prefixed to all messages logged to `syslog`
`imq.log.syslog.logpid`[2]	Boolean	`true`	Log broker process ID with message?
`imq.log.syslog.logconsole`[2]	Boolean	`false`	Write messages to system console if they cannot be sent to `syslog`?
`imq.log.timezone`	String	Local time zone	Time zone for log time stamps
			Possible values are the same as those used by the method `java.util.TimeZone.getTimeZone`.
			Examples:
			■ `GMT`
			■ `GMT-8:00`
			■ `America/LosAngeles`
			■ `Europe/Rome`
			■ `Asia/Tokyo`
`imq.metrics.enabled`	Boolean	`true`	Enable writing of metrics information to Logger?
			Does not affect the production of metrics messages (controlled by `imq.metrics.topic.enabled`).

Table 17–13   (Cont.) Broker Monitoring Properties

Property	Type	Default Value	Description
`imq.metrics.interval`	Integer	`-1`	Time interval, in seconds, at which to write metrics information to Logger
			Does not affect the time interval for production of metrics messages (controlled by `imq.metrics.topic.interval`).
			A value of `-1` denotes an indefinite interval (never write metrics information to Logger).
`imq.metrics.topic.enabled`	Boolean	`true`	Enable production of metrics messages to metric topic destinations?
			If `false`, an attempt to subscribe to a metric topic destination will throw a client-side exception.
`imq.metrics.topic.interval`	Integer	`60`	Time interval, in seconds, at which to produce metrics messages to metric topic destinations
`imq.metrics.topic.persist`	Boolean	`false`	Are metrics messages sent to metric topic destinations persistent?
`imq.metrics.topic.timetolive`	Integer	`300`	Lifetime, in seconds, of metrics messages sent to metric topic destinations
`imq.primaryowner.name`[3]	String	System property `user.name` (user who started the broker)	Name of primary system owner
`imq.primaryowner.contact`[3]	String	System property `user.name` (user who started the broker)	Contact information for primary system owner
`imq.broker.adminDefinedRoles.count`[3]	Integer	None	Number of defined roles
`imq.broker.adminDefinedRoles.nameN`[3]	String	Broker instance name	Name of defined role *N* (where *N* ranges from 0 to .count-1)
			Example:
			`...name0=Stocks JMS Server` `...name1=JMS provider for appserver`

[1] Can be used with `imqcmd update bkr` command

[2] Solaris platform only

[3] Used by JES Monitoring Framework

Cluster Configuration Properties

Table 17–14 lists the configuration properties related to broker clusters.

Table 17–14   Broker Properties for Cluster Configuration

Property	Type	Default Value	Description
imq.cluster.url[1,2]	String	None	URL of `cluster` configuration file, if any
			Examples:
			■ `http://webserver/imq/cluster.properties`
			■ (for a file on a Web server)
			■ `file:/net/mfsserver/imq/cluster.properties`
			■ (for a file on a shared drive)
imq.cluster.hostname[3]	String	None	Host name or IP address for `cluster` connection service
			If specified, overrides `imq.hostname` (see Table 17–1) for the `cluster` connection service. This might be necessary, for instance, if the broker's host computer has more than one interface card installed.
imq.cluster.port[3]	Integer	0	Port number for `cluster` connection service
			A value of `0` specifies that the port number should be allocated dynamically by the Port Mapper. You might need to set a different value, for instance, to specify a static port number for connecting to the broker through a firewall.
imq.cluster.transport[1]	String	tcp	Network transport protocol for `cluster` connection service
			For secure, encrypted message delivery between brokers, set this property to `ssl`.
imq.cluster.ha	Boolean	false	Is broker part of an enhanced (high-availability) cluster?
Additional Properties for Conventional Clusters			
imq.cluster.brokerlist[1,4]	String	None	List of broker addresses belonging to cluster
			The list consists of one or more addresses, separated by commas. Each address specifies the Port Mapper host name and Port Mapper port number of a broker in the cluster, in the form *hostName*:*portNumber*.
			Example:
			■ `host1:3000,host2:8000,ctrlhost`
			Literal IP addresses as host names: You can use a literal IPv4 or IPv6 address as a host name. If you use a literal IPv6 address, its format must conform to RFC2732 (`http://www.ietf.org/rfc/rfc2732.txt`), *Format for Literal IPv6 Addresses in URL's*.
			Note: If set, this property is ignored (and a warning logged) for high-availability clusters; all brokers configured to use the cluster's shared persistent store are automatically recognized as members of the cluster.

Table 17–14   (Cont.) Broker Properties for Cluster Configuration

Property	Type	Default Value	Description
imq.cluster.nomasterbroker[1,4]	Boolean	false	Specifies whether a conventional cluster uses a shared JDBC database store for the cluster configuration change record instead of using a master broker.
			Set to true for a conventional cluster of peer brokers, which uses a shared JDBC store for the cluster's configuration change record.
			When set to true, the imq.cluster.clusterid must be set, and the imq.cluster.sharecc.persist.jdbc.* properties must be configured for accessing the shared JDBC store.
imq.cluster.masterbroker[1,4]	String	None	Port Mapper host name and Port Mapper port number of host on which cluster's master broker (if any) is running.
			The value has the form *hostName:portNumber*, where *hostName* is the Port Mapper host name of the master broker's host and *portNumber* is its Port Mapper port number.
			Example:
			▪ ctrlhost:7676
			Literal IP addresses as host names: You can use a literal IPv4 or IPv6 address as a host name. If you use a literal IPv6 address, its format must conform to RFC2732 (http://www.ietf.org/rfc/rfc2732.txt), *Format for Literal IPv6 Addresses in URL's.*
			Note: enhanced clusters cannot have a master broker. If this property is set for a broker belonging to an enhanced cluster, the broker will log a warning message and ignore the property.
imq.cluster.dynamicChangeMasterBrokerEnabled[1,4]	Boolean	false	Can the master broker for the cluster be changed dynamically; that is, without stopping all the brokers in the cluster?
			If set to true, you can use the imqcmd changemaster command to change the master broker without stopping the brokers in the cluster.
			If set to true, the imq.cluster.masterbroker property cannot be specified on the imqbrokerd command line.
imq.cluster.sharecc.persist.jdbc.connection.retry.max	Integer	60	The maximum number of times a broker will retry a connection.
imq.cluster.sharecc.persist.jdbc.connection.retry.delay	Integer	5000	The amount of time, in milliseconds, the broker waits before retrying a failed connection.
imq.cluster.sharecc.persist.jdbc.dbVendor[1,4]	String	None	Name of database vendor for shared JDBC data store housing the cluster configuration change record:
			▪ db2: DB2
			▪ derby: Java DB (Oracle Corporation)
			▪ oracle: Oracle (Oracle Corporation)
			▪ mysql: MySQL (Oracle Corporation)
			▪ postgresql: postgreSQL

Table 17–14   (Cont.) Broker Properties for Cluster Configuration

Property	Type	Default Value	Description
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.driver`[1,4]	String	per Vendor	Java class name of the JDBC driver, if needed, for connecting to database from vendor *vendorName* for shared JDBC data store housing the cluster configuration change record
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.opendburl`[1,4]	String	None	URL for connecting to existing database from vendor *vendorName* for shared JDBC data store housing the cluster configuration change record
			Applicable when a `java.sql.Driver` is used to connect to database.
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.createdburl`[1,4]	String	None	URL for creating new database from vendor *vendorName* for shared JDBC data store housing the cluster configuration change record
			Applicable for embedded databases, such as Java DB.
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.closedburl`[1,4]	String	None	URL for closing connection to database from vendor *vendorName* for shared JDBC data store housing the cluster configuration change record
			Applicable for some embedded databases, such as Java DB.
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.tableoption`[1,4]	String	None, except for MySQL	Vendor-specific options passed to database from vendor *vendorName* for shared JDBC data store housing the cluster configuration change record when creating the table schema
			For information about this property's use, see the default broker properties file, `default.properties`.
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.connection.retry.regex`*.n*	String	None	This property specifies a regular expression, *n*, to be used to match a `SQLException` message in order to identify a retriable database communication failure.
			There can be 0 or many of these properties (1, 2, 3, ... in consecutive order) specified. The `SQLException` message is matched to these regular expressions in the order of *n* and stops matching when a match is found.
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.user`[1,4]	String	None	User name, if required, for connecting to database from vendor *vendorName* for shared JDBC data store housing the cluster configuration change record
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.needpassword`[1,4]	Boolean	`false`	Does database from vendor *vendorName* for shared JDBC data store housing the cluster configuration change record require a password for broker access?
			If `true`, the `imqbrokerd` and `imqdbmgr` commands will prompt for a password unless the following property, `password` is set.
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.password`[1,4]	String	None	Password, if required, for connecting to database from vendor *vendorName* for shared JDBC data store housing the cluster configuration change record
			This property should only be specified in a password file, as described in Password Files.
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.property.`*propName*[1,4]	String	None	Optional vendor-specific property *propName* for the JDBC driver from vendor *vendorName* for shared JDBC data store housing the cluster configuration change record

Table 17–14   (Cont.) Broker Properties for Cluster Configuration

Property	Type	Default Value	Description
Additional Properties for Enhanced (High-Availability) Clusters			
`imq.cluster.clusterid`[1,5]	String	None	Cluster identifier
			Must be a unique alphanumeric string of no more than n-18 characters, where n is the maximum table name length allowed by the database. No two running clusters may have the same cluster identifier.
			Note: For brokers belonging to a high-availability cluster, this property is used in database table names in place of `imq.brokerid` (see Table 17–1). For conventional cluster of peer brokers, this property is used in the shared database table name for the cluster's configuration change records.
`imq.cluster.ha.takeoverWaitTimeout`[5]	Integer	300	Time in seconds a failed broker attempting to restart waits for an existing takeover activity (from the broker's initial failure) to complete before exiting its restart attempt
`imq.cluster.heartbeat.hostname`[5]	String	None	Host name for heartbeat service
			If specified, overrides `imq.hostname` (see Table 17–1) for the heartbeat service.
`imq.cluster.heartbeat.port`[5]	Integer	`7676`	Port number for heartbeat service
			A value of `0` specifies that the port number should be allocated dynamically by the Port Mapper.
`imq.cluster.heartbeat.interval`[5]	Integer	2	Interval between heartbeats, in seconds
`imq.cluster.heartbeat.threshold`[5]	Integer	3	Number of missed heartbeat intervals after which to invoke monitor service
`imq.cluster.monitor.interval`[5]	Integer	30	Interval, in seconds, at which to update monitor time stamp
			Note: Larger values for this property will reduce the frequency of database access and thus improve overall system performance, but at the cost of slower detection and takeover in the event of broker failure.
`imq.cluster.monitor.threshold`[5]	Integer	2	Number of missed monitor intervals after which to initiate broker takeover

[1]   Must have the same value for all brokers in a cluster.

[2]   Can be used with the `imqcmd update bkr` command.

[3]   Can be specified independently for each broker in a cluster.

[4]   Conventional clusters only

[5]   Enhanced (high-availability) clusters only

Bridge Properties

Table 17–15 lists broker properties related to the bridge service manager. Table 17–16 lists broker properties related specifically to the JMS bridge service, and Table 17–17 lists broker properties related specifically to the STOMP bridge service.

Table 17–15   Broker Properties for the Bridge Service Manager

Property	Type	Default Value	Description
imq.bridge.enabled	Boolean	false	Is the bridge service enabled on this broker?
imq.bridge.activelist	String	None	List of bridges that will be loaded on broker startup.
			The list consists of one or more bridge names, separated by commas. All bridge names for a broker must be unique.
imq.bridge.admin.user	String	None	The Message Queue broker administrative user to be used by the bridge service manager and individual bridges to create ADMIN connections to the broker. For JMS bridges, this user is also used to access the JMS bridge's built-in DMQ destination.
imq.bridge.admin.password	String	None	The password for the imq.bridge.admin.user user.

Table 17–16   Broker Properties for a JMS Bridge Service

Property	Type	Default Value	Description
imq.bridge.*name*.type	String	None	The bridge type of the bridge named *name*. For JMS bridges, specify a value of JMS or jms.
imq.bridge.*name*.xmlurl	String	None	The URL where the XML configuration file for the JMS bridge *name* is stored.
			Examples:
			■ http://webserver/imq/jmsbridge1.config.xml
			■ (for a file on a Web server)
			■ file:/net/fileserver/imq/jmsbridge1.config.xml
			■ (for a file on a shared drive)
imq.bridge.*name*.autostart	Boolean	true	Should the JMS bridge *name* be automatically started when the broker is started?
imq.bridge.*name*.logfile.limit	Integer	0	The approximate maximum number of bytes the JMS bridge *name* writes to any one log file.
			A value of 0 (zero) indicates that there is no maximum limit.
imq.bridge.*name*.logfile.count	Integer	1	The number of log files the JMS bridge *name* cycles through.
imq.bridge.tm.props imq.bridge.*name*.tm.props	String	None	Each of these properties specifies a list of key-value pairs for the built-in transaction coordinator for the JMS bridge *name*.
			The list consists of one or more *key=value* pairs separated by commas.
			When the imq.persist.store is file, the built-in transaction coordinator supports these keys: txlogSize, txlogSync, and txlogMmap.
			If the same key appears in both properties, the value specified in imq.bridge.*name*.tm.props takes precedence.

Table 17–17 Broker Properties for the STOMP Bridge Service

Property	Type	Default Value	Description
`imq.bridge.stomp.hostname`	String	None	Host name or IP address for the STOMP bridge service
			If specified, overrides `imq.hostname` (see Table 17–1) for the STOMP bridge service.
`imq.bridge.stomp.tcp.enabled`	Boolean	`true`	Does the STOMP bridge accept TCP connections?
`imq.bridge.stomp.tcp.port`	Integer	`7672`	The port on which the STOMP bridge listens for TCP connections, provided that `imq.bridge.stomp.tcp.enabled` is `true`.
`imq.bridge.stomp.tls.enabled`	Boolean	`false`	Does the STOMP bridge accept SSL/TLS connections?
			If `true`, a keystore must be created using the `imqkeytool` utility before starting the broker.
`imq.bridge.stomp.tls.port`	Integer	`7673`	The port on which the STOMP bridge listens for SSL/TLS connections, provided that `imq.bridge.stomp.tls.enabled` is `true`.
`imq.bridge.stomp.tls.requireClientAuth`	Boolean	`false`	Do SSL/TLS connections require client authentication?
`imq.bridge.stomp.consumerFlowLimit`	Integer	`1000`	The maximum number of unacknowledged messages that the STOMP bridge will deliver on a transacted STOMP subscription. The STOMP client must then acknowledge the messages and commit the transaction.
`imq.bridge.stomp.messageTransformer`	String	None	The fully qualified class name of a class that extends the Message Queue bridge `MessageTransformer` abstract class by implementing the `transform()` method. Place this class under the `IMQ_HOME/lib/ext.` directory
`imq.bridge.stomp.logfile.limit`	Integer	`0`	The approximate maximum number of bytes the STOMP bridge writes to any one log file.
			A value of `0` (zero) indicates that there is no maximum limit.
`imq.bridge.stomp.logfile.count`	Integer	`1`	The number of log files the STOMP bridge cycles through.

JMX Properties

The broker properties listed in Table 17–18 support the use of the Java Management Extensions (JMX) application programming interface by Java applications. The JMX API is used to configure and monitor broker resources.

These JMX-related properties can be set in the broker's instance configuration file (`config.properties`) or at broker startup with the `-D` option of the Broker utility (`imqbrokerd`). None of these properties can be set dynamically with the Command utility (`imqcmd`).

In addition, some of these properties (`imq.jmx.rmiregistry.start`, `imq.jmx.rmiregistry.use`, `imq.jmx.rmiregistry.port`) can be set with corresponding Broker utility `imqbrokerd` options described in Table 16–1.

See JMX Support for further information on administrative support of JMX clients.

Table 17–18   Broker Properties for JMX Support

Property	Type	Default Value	Description
`imq.jmx.connector.activelist`	String	`jmxrmi`	Names of JMX connectors to be activated at broker startup, separated by commas
`imq.jmx.connector.`*RMIconnectorName*`.url` `path`	String	Shown in next column	*urlpath* component of JMX service URL for connector *connectorName*
			Useful in cases where an RMI registry is being used and the JMX service URL path must be set explicitly (such as when a shared external RMI registry is used). See The JMX Service URL.
			Default:
			`/jndi/rmi://`*brokerHost*`:`*rmiPort* `/`*brokerHost*`/`*brokerPort*`/`*connectorNam* *e*
`imq.jmx.connector.`*RMIconnectorName*`.por` `t`	Integer	None: the port is dynamically allocated	Port number of JMX connector
			Used to specify a static/known JMX connector port, typically in cases where a JMX client is accessing the broker's MBean server through a firewall. See JMX Connections Through a Firewall.
`imq.jmx.connector.`*RMIconnectorName*`.use` `SSL`	Boolean	`false`	Use Secure Socket Layer (SSL) for connector *connectorName*?
			This property is set to `true` for the `ssljmxrmi` connector.
`imq.jmx.connector.`*RMIconnectorName*`.bro` `kerHostTrusted`	Boolean	`false`	Trust any certificate presented by broker for connector *connectorName*?
			Applies only when `imq.jmx.connector.`*connectorName*`.useS` `SL` is `true`.
			If `false`, the JMX client runtime will validate all certificates presented to it. Validation will fail if the signer of the certificate is not in the client's trust store.
			If `true`, validation of certificates is skipped. This can be useful, for instance, during software testing when a self-signed certificate is used.

Table 17–18   (Cont.) Broker Properties for JMX Support

Property	Type	Default Value	Description
imq.jmx.rmiregistry.start	Boolean	false	Start RMI registry at broker startup?
			If true, the broker will start an RMI registry at the port specified by imq.jmx.rmiregistry.port and use the regsitry to store the JMX connector stub. (The value of imq.jmx.rmiregistry.use is ignored in this case.)
			For convenience, this property can also be set at broker startup with the -startRmiRegistry option of imqbrokerd.
imq.jmx.rmiregistry.use	Boolean	false	Use an existing RMI registry?
			Applies only if imq.jmx.rmiregistry.start is false.
			If true, the broker will use an existing RMI registry on the local host at the port specified by imq.jmx.rmiregistry.port to store the JMX connector stub. The existing RMI registry must already be running at broker startup.
			For convenience, this property can also be set at broker startup with the -useRmiRegistry option of imqbrokerd.
imq.jmx.rmiregistry.port	Integer	1099	Port number of RMI registry
			Applies only if imq.jmx.rmiregistry.start is true or imq.jmx.rmiregistry.use is true.
			This port number will be included in the URL path of the JMX service URL.
			For convenience, this property can also be set at broker startup with the -rmiRegistryPort option of imqbrokerd.

Alphabetical List of Broker Properties

Alphabetical List of Broker Properties is an alphabetical list of broker configuration properties, with cross-references to the relevant tables in this chapter.

Table 17–19   Alphabetical List of Broker Properties

Property	Table
imq.accesscontrol.enabled	Table 17–9
imq.accesscontrol.type	Table 17–9
imq.accesscontrol.file.filename	Table 17–9
imq.admincreate.topic.sharedConsumerFlowLimit	Table 17–4
imq.audit.bsm.disabled	Table 17–9
imq.audit.enabled	Table 17–9
imq.authentication.basic.user_repository	Table 17–9
imq.authentication.client.response.timeout	Table 17–9

Table 17–19   (Cont.) Alphabetical List of Broker Properties

Property	Table
`imq.authentication.type`	Table 17–9
`imq.autocreate.destination.isLocalOnly`	Table 17–3
`imq.autocreate.destination.limitBehavior`	Table 17–3
`imq.autocreate.destination.maxBytesPerMsg`	Table 17–3
`imq.autocreate.destination.maxNumMsgs`	Table 17–3
`imq.autocreate.destination.maxNumProducers`	Table 17–3
`imq.autocreate.destination.maxTotalMsgBytes`	Table 17–3
`imq.autocreate.destination.useDMQ`	Table 17–3
`imq.autocreate.queue`	Table 17–3
`imq.autocreate.queue.consumerFlowLimit`	Table 17–3
`imq.autocreate.queue.localDeliveryPreferred`	Table 17–3
`imq.autocreate.queue.maxNumActiveConsumers`	Table 17–3
`imq.autocreate.queue.maxNumBackupConsumers`	Table 17–3
`imq.autocreate.reaptime`	Table 17–3
`imq.autocreate.topic`	Table 17–3
`imq.autocreate.topic.consumerFlowLimit`	Table 17–3
`imq.autocreate.topic.sharedConsumerFlowLimit`	Table 17–3
`imq.broker.adminDefinedRoles.count`	Table 17–13
`imq.broker.adminDefinedRoles.name`*n*	Table 17–13
`imq.brokerid`	Table 17–1
`imq.bridge.activelist`	Table 17–15
`imq.bridge.admin.password`	Table 17–15
`imq.bridge.admin.user`	Table 17–15
`imq.bridge.enabled`	Table 17–15
`imq.bridge.`*name*`.autostart`	Table 17–16
`imq.bridge.`*name*`.logfile.count`	Table 17–16
`imq.bridge.`*name*`.logfile.limit`	Table 17–16
`imq.bridge.`*name*`.tm.props`	Table 17–16
`imq.bridge.`*name*`.type`	Table 17–16
`imq.bridge.`*name*`.xmlurl`	Table 17–16
`imq.bridge.stomp.consumerFlowLimit`	Table 17–17
`imq.bridge.stomp.logfile.count`	Table 17–17
`imq.bridge.stomp.logfile.limit`	Table 17–17
`imq.bridge.stomp.messageTransformer`	Table 17–17
`imq.bridge.stomp.tcp.enabled`	Table 17–17
`imq.bridge.stomp.tcp.port`	Table 17–17
`imq.bridge.stomp.tls.enabled`	Table 17–17

Table 17–19 (Cont.) Alphabetical List of Broker Properties

Property	Table
`imq.bridge.stomp.tls.port`	Table 17–17
`imq.bridge.stomp.tls.requireClientAuth`	Table 17–17
`imq.bridge.tm.props`	Table 17–16
`imq.cluster.brokerlist`	Table 17–14
`imq.cluster.clusterid`	Table 17–14
`imq.cluster.dynamicChangeMasterBrokerEnabled`	Table 17–14
`imq.cluster.ha`	Table 17–14
`imq.cluster.heartbeat.hostname`	Table 17–14
`imq.cluster.heartbeat.interval`	Table 17–14
`imq.cluster.heartbeat.port`	Table 17–14
`imq.cluster.heartbeat.threshold`	Table 17–14
`imq.cluster.hostname`	Table 17–14
`imq.cluster.nomasterbroker`	Table 17–14
`imq.cluster.masterbroker`	Table 17–14
`imq.cluster.monitor.interval`	Table 17–14
`imq.cluster.monitor.threshold`	Table 17–14
`imq.cluster.port`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.connection.retry.delay`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.connection.retry.max`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.dbVendor`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.connection.retry.` `regex.`*n*	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.driver`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.closedburl`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.createdburl`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.needpassword`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.opendburl`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.password`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.property.`*propName* *e*	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.tableoption`	Table 17–14
`imq.cluster.sharecc.persist.jdbc.`*vendorName*`.user`	Table 17–14
`imq.cluster.transport`	Table 17–14
`imq.cluster.url`	Table 17–14
`imq.destination.DMQ.truncateBody`	Table 17–2
`imq.destination.logDeadMsgs`	Table 17–13
`imq.hostname`	Table 17–1
`imq.imqcmd.password`	Table 17–9

Table 17–19   (Cont.) Alphabetical List of Broker Properties

Property	Table
`imq.jmx.connector.activelist`	Table 17–18
`imq.jmx.connector.`*RMIconnectorName*`.brokerHostTrusted`	Table 17–18
`imq.jmx.connector.`*RMIconnectorName*`.port`	Table 17–18
`imq.jmx.connector.`*RMIconnectorName*`.urlpath`	Table 17–18
`imq.jmx.connector.`*RMIconnectorName*`.useSSL`	Table 17–18
`imq.jmx.rmiregistry.port`	Table 17–18
`imq.jmx.rmiregistry.start`	Table 17–18
`imq.jmx.rmiregistry.use`	Table 17–18
`imq.keystore.file.dirpath`	Table 17–9
`imq.keystore.file.name`	Table 17–9
`imq.keystore.password`	Table 17–9
`imq.keystore.`*propertyName*	Table 17–9
`imq.log.console.output`	Table 17–13
`imq.log.console.stream`	Table 17–13
`imq.log.file.dirpath`	Table 17–13
`imq.log.file.filename`	Table 17–13
`imq.log.file.output`	Table 17–13
`imq.log.file.rolloverbytes`	Table 17–13
`imq.log.file.rolloversecs`	Table 17–13
`imq.log.level`	Table 17–13
`imq.log.syslog.facility`	Table 17–13
`imq.log.syslog.identity`	Table 17–13
`imq.log.syslog.logconsole`	Table 17–13
`imq.log.syslog.logpid`	Table 17–13
`imq.log.syslog.output`	Table 17–13
`imq.log.timezone`	Table 17–13
`imq.message.expiration.interval`	Table 17–2
`imq.message.max_size`	Table 17–2
`imq.metrics.enabled`	Table 17–13
`imq.metrics.interval`	Table 17–13
`imq.metrics.topic.enabled`	Table 17–13
`imq.metrics.topic.interval`	Table 17–13
`imq.metrics.topic.persist`	Table 17–13
`imq.metrics.topic.timetolive`	Table 17–13
`imq.passfile.dirpath`	Table 17–9
`imq.passfile.enabled`	Table 17–9
`imq.passfile.name`	Table 17–9

Table 17–19   (Cont.) Alphabetical List of Broker Properties

Property	Table
imq.persist.file.destination.message.filepool.limit	Table 17–6
imq.persist.file.message.cleanup	Table 17–6
imq.persist.file.message.filepool.cleanratio	Table 17–6
imq.persist.file.message.max_record_size	Table 17–6
imq.persist.file.sync.enabled	Table 17–6
imq.persist.file.transaction.memorymappedfile.enabled	Table 17–6
imq.persist.jdbc.dbVendor	Table 17–8
imq.persist.jdbc.connection.retry.delay	Table 17–8
imq.persist.jdbc.connection.retry.max	Table 17–8
imq.persist.jdbc.*vendorName*.connection.retry.regex.*n*	Table 17–8
imq.persist.jdbc.*vendorName*.closedburl	Table 17–8
imq.persist.jdbc.*vendorName*.createdburl	Table 17–8
imq.persist.jdbc.*vendorName*.driver	Table 17–8
imq.persist.jdbc.*vendorName*.needpassword	Table 17–8
imq.persist.jdbc.*vendorName*.opendburl	Table 17–8
imq.persist.jdbc.*vendorName*.password	Table 17–8
imq.persist.jdbc.*vendorName*.property.*propName*	Table 17–8
imq.persist.jdbc.*vendorName*.user	Table 17–8
imq.persist.store	Table 17–5
imq.ping.interval	Table 17–1
imq.portmapper.backlog	Table 17–1
imq.portmapper.hostname	Table 17–1
imq.portmapper.port	Table 17–1
imq.primaryowner.contact	Table 17–13
imq.primaryowner.name	Table 17–13
imq.*resourceState*.count	Table 17–2
imq.*resourceState*.threshold	Table 17–2
imq.service.activelist	Table 17–1
imq.*serviceName*.accesscontrol.enabled	Table 17–9
imq.*serviceName*.accesscontrol.file.filename	Table 17–9
imq.*serviceName*.authentication.type	Table 17–9
imq.*serviceName*.max_threads	Table 17–1
imq.*serviceName*.min_threads	Table 17–1
imq.*serviceName*.*protocolType*.hostname	Table 17–1
imq.*serviceName*.*protocolType*.port	Table 17–1
imq.*serviceName*.threadpool_model	Table 17–1
imq.shared.connectionMonitor_limit	Table 17–1

Table 17–19   (Cont.) Alphabetical List of Broker Properties

Property	Table
`imq.system.max_count`	Table 17–2
`imq.system.max_size`	Table 17–2
`imq.transaction.autorollback`	Table 17–2
`imq.user_repository.file.dirpath`	Table 17–10
`imq.user_repository.file.filename`	Table 17–10
`imq.user_repository.ldap.base`	Table 17–11
`imq.user_repository.ldap.gidattr`	Table 17–11
`imq.user_repository.ldap.grpbase`	Table 17–11
`imq.user_repository.ldap.grpfilter`	Table 17–11
`imq.user_repository.ldap.grpsearch`	Table 17–11
`imq.user_repository.ldap.memattr`	Table 17–11
`imq.user_repository.ldap.password`	Table 17–11
`imq.user_repository.ldap.principal`	Table 17–11
`imq.user_repository.ldap.`*propertyName*	Table 17–11
`imq.user_repository.ldap.server`	Table 17–11
`imq.user_repository.ldap.ssl.enabled`	Table 17–11
`imq.user_repository.ldap.ssl.socketfactory`	Table 17–11
`imq.user_repository.ldap.timeout`	Table 17–11
`imq.user_repository.ldap.uidattr`	Table 17–11
`imq.user_repository.ldap.usrfilter`	Table 17–11
`imq.user_repository.jaas.name`	Table 17–12
`imq.user_repository.jaas.userPrincipalClass`	Table 17–12
`imq.user_repository.jaas.groupPrincipalClass`	Table 17–12

18

Physical Destination Property Reference

This chapter provides reference information about configuration properties for physical destinations.

Physical Destination Properties

Table 18–1 lists the configuration properties for physical destinations. These properties can be set when creating or updating a physical destination. For auto-created destinations, you set default values in the broker's instance configuration file (see Table 17–3).

Table 18–1   Physical Destination Properties

Property	Type	Default Value	Description
maxNumMsgs[1]	Integer	-1	Maximum number of unconsumed messages
			A value of -1 denotes an unlimited number of messages.
			For the dead message queue, the default value is 1000.
			Note: When flow control is in effect (limitBehavior = FLOW_CONTROL), it is possible for the specified message limit to be exceeded because the broker cannot react quickly enough to stop the flow of incoming messages. In such cases, the value specified for maxNumMsgs serves as merely a hint for the broker rather than a strictly enforced limit. However, if the number of unconsumed messages would exceed imq.system.max_count, the broker generates a ResourceAllocationException indicating that the destination is full and rejecting new messages.
maxBytesPerMsg	String	-1	Maximum size, in bytes, of any single message
			Rejection of a persistent message is reported to the producing client with an exception; no notification is sent for nonpersistent messages.
			The value may be expressed in bytes, kilobytes, or megabytes, using the following suffixes:
			■ b: Bytes
			■ k: Kilobytes (1024 bytes)
			■ m: Megabytes (1024 × 1024 = 1,048,576 bytes)
			An unsuffixed value is expressed in bytes; a value of -1 denotes an unlimited message size.

Table 18–1   (Cont.) Physical Destination Properties

Property	Type	Default Value	Description
			Examples:
			■ `1600`: 1600 bytes
			■ `1600b`: 1600 bytes
			■ `16k`: 16 kilobytes (= 16,384 bytes)
			■ `16m`: 16 megabytes (= 16,777,216 bytes)
			■ `-1`: No limit
`maxTotalMsgBytes`[1]	String	`-1`	Maximum total memory, in bytes, for unconsumed messages
			The syntax is the same as for `maxBytesPerMsg` (see above).
			For the dead message queue, the default value is `10m`.
`limitBehavior`	String	`REJECT_NEWEST`	Broker behavior when memory-limit threshold reached:
			■ `FLOW_CONTROL`: Slow down producers
			■ `REMOVE_OLDEST`: Throw out oldest messages
			■ `REMOVE_LOW_PRIORITY`: Throw out lowest-priority messages according to age; no notification to producing client
			■ `REJECT_NEWEST`: Reject newest messages; notify producing client with an exception only if message is persistent
			When `FLOW_CONTROL` is specified, it is still possible for the number of messages to exceed `imq.system.max_count`. In this situation, the broker generates a `ResourceAllocationException` indicating that the destination is full and rejecting new messages.
			If the value is `REMOVE_OLDEST` or `REMOVE_LOW_PRIORITY` and the `useDMQ` property is `true`, excess messages are moved to the dead message queue. For the dead message queue itself, the default limit behavior is `REMOVE_OLDEST` and cannot be set to `FLOW_CONTROL`.
`maxNumProducers`[2]	Integer	`100`	Maximum number of message producers for destination
			When this limit is reached, no new producers can be created. A value of `-1` denotes an unlimited number of producers.
`maxNumActiveConsumers`[3]	Integer	`-1`	Maximum number of active message consumers in load-balanced delivery from queue destination
			A value of `-1` denotes an unlimited number of consumers.
			This property used mostly in cases where message order is important and you want to provide backup consumers in case the principal consumer of a queue fails. If message order is not important, then you would simply use multiple consumers to provide for scalability and availability.
`maxNumBackupConsumers`[3]	Integer	`0`	Maximum number of backup message consumers in load-balanced delivery from queue destination
			A value of `-1` denotes an unlimited number of consumers.

Table 18–1   (Cont.) Physical Destination Properties

Property	Type	Default Value	Description
consumerFlowLimit	Integer	1000	Maximum number of messages delivered to a consumer in a single batch
			In load-balanced queue delivery, this is the initial number of queued messages routed to an active consumer before load balancing begins.
			The client runtime can override this limit by specifying a lower value on the connection factory object.
			A value of 0 or -1 denotes an unlimited number of messages.
			Not used when the JMS resource adapter, jmsra, is used to consume messages in a GlassFish Server cluster.
isLocalOnly[2]	Boolean	false	Local delivery only?
			This property applies only to destinations in broker clusters, and cannot be changed once the destination has been created. If true, the destination is not replicated on other brokers and is limited to delivering messages only to local consumers (those connected to the broker on which the destination is created).
localDeliveryPreferred[2],[3]	Boolean	false	Local delivery preferred?
			This property applies only to load-balanced queue delivery in broker clusters. If true, messages will be delivered to remote consumers only if there are no consumers on the local broker; the destination must not be restricted to local-only delivery (isLocalOnly must be false).
useDMQ[2]	Boolean	true	Send dead messages to dead message queue?
			If false, dead messages will simply be discarded.

Table 18–1   (Cont.) Physical Destination Properties

Property	Type	Default Value	Description
validateXMLSchemaEnabled [4]	Boolean	false	XML schema validation is enabled?
			When XML validation is enabled, the Message Queue client runtime will attempt to validate an XML message against the specified XSDs (or against the DTD, if no XSD is specified) before sending it to the broker. If the specified schema cannot be located or the message cannot be validated, the message is not sent, and an exception is thrown. Client applications using this feature should use JRE 1.5 or above.
			If set to false or not set, then XML schema validation is not enabled for the destination.
XMLSchemaURIList[4]	String	null	Space separated list of XML schema document (XSD) URI strings
			The URIs point to the location of one or more XSDs to use for XML schema validation, if enabled.
			Use double quotes around this value if multiple URIs are specified.
			Example:
			"http://foo/flap.xsd http://test.com/test.xsd"
			If this property is not set or null and XML validation is enabled, XML validation is performed using a DTD specified in the XML document.
			if an XSD is changed, as a result of changing application requirements, all client applications producing XML messages based on the changed XSD must reconnect to the broker.
reloadXMLSchemaOnFailure[4]	Boolean	false	Reload XML schema on failure enabled?
			If set to true and XML validation fails, then the Message Queue client runtime will attempt to reload the XSD before attempting again to validate a message. The client runtime will throw an exception if the validation fails using the reloaded SXD.
			If set to false or not set, then the schema is not reloaded if validation fails.

[1]   In a cluster environment, applies to each individual instance of a destination rather than collectively to all instances in the cluster

[2]   Does not apply to dead message queue

[3]   Queue destinations only

[4]   This property should be set when a destination is inactive: when it has no consumers or producers and when there are no messages in the destination. Otherwise the producer must reconnect.

19

Administered Object Attribute Reference

This chapter provides reference information about the attributes of administered objects. It consists of the following sections:

- Connection Factory Attributes
- Destination Attributes

Connection Factory Attributes

The attributes of a connection factory object are grouped into categories described in the following sections below:

- Connection Handling
- Client Identification
- Reliability and Flow Control
- Queue Browser and Server Sessions
- Standard Message Properties
- Message Header Overrides

Connection Handling

Table 19–1 lists the connection factory attributes for connection handling.

Table 19–1   Connection Factory Attributes for Connection Handling

Attribute	Type	Default Value	Description
imqAddressList	String	An existing Message Queue 3.0 address, if any; if none, the first entry in Table 19–2	List of broker addresses The list consists of one or more addresses, separated by commas. Each address specifies (or implies) the host name, port number, and connection service for a broker instance to which the client can connect. Address syntax varies depending on the connection service and port assignment method; see below for details. **Literal IP addresses as host names:** You can use a literal IPv4 or IPv6 address as a host name. If you use a literal IPv6 address, its format must conform to RFC2732 (http://www.ietf.org/rfc/rfc2732.txt), *Format for Literal IPv6 Addresses in URL's.* **Note:** In an enhanced broker cluster, the value of this attribute is updated dynamically as brokers enter and leave the cluster, so that it always reflects the cluster's current membership.
imqAddressListBehavior	String	PRIORITY	Order in which to attempt connection to broker addresses: ■ PRIORITY: Order specified in address list ■ RANDOM: Random order **Note:** If many clients share the same connection factory, specify random connection order to prevent them from all attempting to connect to the same address.
imqAddressListIterations	Integer	1	Number of times to iterate through address list attempting to establish or reestablish a connection A value of -1 denotes an unlimited number of iterations. **Note:** In the event of broker failure in an enhanced broker cluster, this attribute is ignored and the Message Queue client runtime iterates through the address list indefinitely until it succeeds in reconnecting to a takeover broker. The effect is equivalent to an imqAddressListIterations value of -1, overriding any other explicit or default setting of this attribute. The only way for a client application to avoid this behavior is to close the connection explicitly on broker failure.
imqPingInterval	Integer	30	Interval, in seconds, at which to test connection between client and broker A value of 0 or -1 disables periodic testing of the connection.
imqReconnectEnabled	Boolean	false	Attempt to reestablish a lost connection? **Note:** In the event of broker failure in an enhanced broker cluster, this attribute is ignored and automatic reconnection is always attempted. The effect is equivalent to an imqReconnectEnabled value of true, overriding any other explicit or default setting of this attribute. The only way for a client application to avoid this behavior is to close the connection explicitly on broker failure.

Table 19–1 (Cont.) Connection Factory Attributes for Connection Handling

Attribute	Type	Default Value	Description
imqReconnectAttempts	Integer	0	Number of times to attempt connection (or reconnection) to each address in address list before moving on to next
			A value of -1 denotes an unlimited number of connection attempts: attempt repeatedly to connect to first address until successful. For example, in an enhanced broker cluster, this value will allow for connection to the failover broker.
imqReconnectInterval	Long integer	3000	Interval, in milliseconds, between reconnection attempts
			This value applies both for successive attempts on a given address and for successive addresses in the list.
			Note: Too small a value may give the broker insufficient recovery time; too large a value may cause unacceptable connection delays.
imqSocketConnectTimeout	Long integer	0	Socket timeout, in milliseconds, used when a TCP connection is made to the broker
			This value applies when connecting to the port mapper as well as when connecting to the required service.
			The timeout value 0 (zero) denotes an infinite timeout, in which case the connection will block until it is established or an error occurs.
imqSSLIsHostTrusted	Boolean	false	Trust any certificate presented by broker?
			If false, the Message Queue client runtime will validate all certificates presented to it. Validation will fail if the signer of the certificate is not in the client's trust store.
			If true, validation of certificates is skipped. This can be useful, for instance, during software testing when a self-signed certificate is used.
			NOTE: To use signed certificates from a certification authority, set this attribute to false.

The value of the imqAddressList attribute is a comma-separated string specifying one or more broker addresses to which to connect. The general syntax for each address is as follows:

scheme:// *address*

where *scheme* identifies one of the addressing schemes shown in the first column of Table 19–2 and *address* denotes the broker address itself. The exact syntax for specifying the address depends on the addressing scheme, as shown in the last column of the table.

Table 19–2   Message Broker Addressing Schemes

Scheme	Service	Syntax	Description
mq	jms or ssljms	[*hostName*][:*portNumber*][/*serviceName*]	Assign port dynamically for jms or ssljms connection service The address list entry specifies the host name and port number for the Message Queue Port Mapper. The Port Mapper itself dynamically assigns a port to be used for the connection. **Default values:** ■ *hostName* = localhost ■ *portNumber* = 7676 ■ *serviceName* = jms For the ssljms connection service, all variables must be specified explicitly. **Literal IP addresses as host names:** You can use a literal IPv4 or IPv6 address as a host name. If you use a literal IPv6 address, its format must conform to RFC2732 (http://www.ietf.org/rfc/rfc2732.txt), *Format for Literal IPv6 Addresses in URL's*.
mqtcp	jms	*hostName*:*portNumber*/jms	Connect to specified port using jms connection service Bypasses the Port Mapper and makes a TCP connection directly to the specified host name and port number.
mqssl	ssljms	*hostName*:*portNumber*/ssljms	Connect to specified port using ssljms connection service Bypasses the Port Mapper and makes a secure SSL connection directly to the specified host name and port number.
http	httpjms	http://*hostName*:*portNumber*/*contextRoot*/tunnel If multiple broker instances use the same tunnel servlet, the following syntax connects to a specific broker instance rather than a randomly selected one: http://*hostName*:*portNumber*/*contextRoot*/tunnel? ■ ServerName=*hostName*:*instanceName*	Connect to specified port using httpjms connection service Makes an HTTP connection to a Message Queue tunnel servlet at the specified URL. The broker must be configured to access the HTTP tunnel servlet.
https	httpsjms	https://*hostName*:*portNumber*/*contextRoot*/tunnel If multiple broker instances use the same tunnel servlet, the following syntax connects to a specific broker instance rather than a randomly selected one: https://*hostName*:*portNumber*/*contextRoot*/tunnel? ■ ServerName=*hostName*:*instanceName*	Connect to specified port using httpsjms connection service Makes a secure HTTPS connection to a Message Queue tunnel servlet at the specified URL. The broker must be configured to access the HTTPS tunnel servlet.

Table 19–3   Message Broker Address Examples

Service	Broker Host	Port	Example Address
Not specified	Not specified	Not specified	No address (`mq://localHost:7676/jms`)
Not specified	Specified host	Not specified	`myBkrHost` (`mq://myBkrHost:7676/jms`)
Not specified	Not specified	Specified Port Mapper port	`1012` (`mq://localHost:1012/jms`)
`ssljms`	Local host	Standard Port Mapper port	`mq://localHost:7676/ssljms`
`ssljms`	Specified host	Standard Port Mapper port	`mq://myBkrHost:7676/ssljms`
`ssljms`	Specified host	Specified Port Mapper port	`mq://myBkrHost:1012/ssljms`
`jms`	Local host	Specified service port	`mqtcp://localhost:1032/jms`
`ssljms`	Specified host	Specified service port	`mqssl://myBkrHost:1034/ssljms`
`httpjms`	Not applicable	Not applicable	`http://websrvr1:8085/imq/tunnel`
`httpsjms`	Not applicable	Not applicable	`https://websrvr2:8090/imq/tunnel`

Client Identification

Table 19–4 lists the connection factory attributes for client identification.

Table 19–4   Connection Factory Attributes for Client Identification

Attribute	Type	Default Value	Description
`imqDefaultUsername`	String	`guest`	Default user name for authenticating with broker
`imqDefaultPassword`	String	`guest`	Default password for authenticating with broker
`imqConfiguredClientID`	String	`null`	Administratively configured client identifier
`imqDisableSetClientID`	Boolean	`false`	Prevent client from changing client identifier using `setClientID` method?

Reliability and Flow Control

Table 19–5 lists the connection factory attributes for reliability and flow control.

Table 19–5   Connection Factory Attributes for Reliability and Flow Control

Attribute	Type	Default Value	Description
`imqAbortOnPingAckTimeout`	Boolean	`false`	Should the connection to the broker be aborted when `imqPingAckTimeout` milliseconds have passed?
			If the connection is aborted, the Message Queue Client Runtime operates as though the connection to the broker were broken.
`imqAckTimeout`	String	`0`	Maximum time, in milliseconds, to wait for broker acknowledgment before throwing an exception
			A value of `0` denotes no timeout (wait indefinitely).
			Note: In some situations, too low a value can cause premature timeout: for example, initial authentication of a user against an LDAP user repository using a secure (SSL) connection can take more than 30 seconds.
`imqAsyncSendCompletionWaitTimeout`	Long	180000	The minimum amount of time, in milliseconds, that a MQ client waits for an asynchronous send to complete before calling `CompletionListener.onException`.

Table 19–5   (Cont.) Connection Factory Attributes for Reliability and Flow Control

Attribute	Type	Default Value	Description
imqConnectionFlowCount	Integer	100	Number of payload messages in a metered batch
			Delivery of payload messages to the client is temporarily suspended after this number of messages, allowing any accumulated control messages to be delivered. Payload message delivery is resumed on notification by the client runtime, and continues until the count is again reached.
			A value of 0 disables metering of message delivery and may cause Message Queue control messages to be blocked by heavy payload message traffic.
imqConnectionFlowLimitEnabled	Boolean	false	Limit message flow at connection level?
imqConnectionFlowLimit	Integer	1000	Maximum number of messages per connection to deliver and buffer for consumption
			Message delivery on a connection stops when the number of unconsumed payload messages pending (subject to flow metering governed by imqConnectionFlowCount) exceeds this limit. Delivery resumes only when the number of pending messages falls below the limit. This prevents the client from being overwhelmed with pending messages that might cause it to run out of memory.
			This attribute is ignored if imqConnectionFlowLimitEnabled is false.
imqConsumerFlowLimitPrefetch	Boolean	true	Is message prefetching and buffering as specified by imqConsumerFlowLimit and imqConsumerFlowThreshold enabled?
			When set to false, message prefetching and buffering is disabled, and each consumer is delivered one message at a time, which can give rise to a number of performance issues.
			This property should be set to false only when business logic demands that each consumer have only one message at a time.

Table 19–5   (Cont.) Connection Factory Attributes for Reliability and Flow Control

Attribute	Type	Default Value	Description
`imqConsumerFlowLimit`	Integer	`1000`	Maximum number of messages per consumer to deliver and buffer for consumption
			Message delivery to a given consumer stops when the number of unconsumed payload messages pending for that consumer exceeds this limit. Delivery resumes only when the number of pending messages for the consumer falls below the percentage specified by `imqConsumerFlowThreshold`. This can be used to improve load balancing among multiple consumers and prevent any single consumer from starving others on the same connection.
			A value of `0` or `-1` denotes an unlimited number of messages.
			This limit can be overridden by a lower value set for a queue's own `consumerFlowLimit` attribute (see Physical Destination Property Reference). Note also that message delivery to all consumers on a connection is subject to the overall limit specified by `imqConnectionFlowLimit`.
`imqConsumerFlowThreshold`	Integer	`50`	Number of messages per consumer buffered in the client runtime, as a percentage of `imqConsumerFlowLimit`, below which to resume message delivery
`imqPingAckTimeout`	Integer	`0`	Maximum time, in milliseconds, to wait for a ping reply or any data sent from the broker since its last ping reply or data sent.
			A value of `0` denotes no timeout (wait indefinitely).
			This value is ignored if the `imqAbortOnPingAckTimeout` attribute is set to `false`.

Queue Browser and Server Sessions

Table 19–6 lists the connection factory attributes for queue browsing and server sessions.

Table 19–6   Connection Factory Attributes for Queue Browser and Server Sessions

Attribute	Type	Default Value	Description
imqQueueBrowserMaxMessagesPerRetrieve	Integer	1000	Maximum number of messages to retrieve at one time when browsing contents of a queue destination
			Note: This attribute does not affect the total number of messages browsed, only the way they are chunked for delivery to the client runtime (fewer but larger chunks or more but smaller ones). The client application will always receive all messages in the queue. Changing the attribute's value may affect performance, but will not affect the total amount of data retrieved.
imqQueueBrowserRetrieveTimeout	Long integer	60000	Maximum time, in milliseconds, to wait to retrieve messages, when browsing contents of a queue destination, before throwing an exception
imqLoadMaxToServerSession	Boolean	true	Load up to maximum number of messages into a server session?
			If `false`, the client will load only a single message at a time.
			This attribute applies only to JMS application server facilities.

Standard Message Properties

The connection factory attributes listed in Table 19–7 control whether the Message Queue client runtime sets certain standard message properties defined in the *Java Message Service Specification.*

Table 19–7   Connection Factory Attributes for Standard Message Properties

Property	Type	Default Value	Description
imqSetJMSXUserID	Boolean	false	Set `JMSXUserID` property (identity of user sending message) for produced messages?
imqSetJMSXAppID	Boolean	false	Set `JMSXAppID` property (identity of application sending message) for produced messages?
imqSetJMSXProducerTXID	Boolean	false	Set `JMSXProducerTXID` property (transaction identifier of transaction within which message was produced) for produced messages?
imqSetJMSXConsumerTXID	Boolean	false	Set `JMSXConsumerTXID` property (transaction identifier of transaction within which message was consumed) for consumed messages?
imqSetJMSXRcvTimestamp	Boolean	false	Set `JMSXRcvTimestamp` property (time message delivered to consumer) for consumed messages?

Message Header Overrides

Table 19–8 lists the connection factory attributes for overriding JMS message header fields.

Table 19–8   Connection Factory Attributes for Message Header Overrides

Attribute	Type	Default Value	Description
`imqOverrideJMSDeliveryMode`	Boolean	`false`	Allow client-set delivery mode to be overridden?
`imqJMSDeliveryMode`	Integer	`2`	Overriding value of delivery mode: `1` Nonpersistent `2` Persistent
`imqOverrideJMSExpiration`	Boolean	`false`	Allow client-set expiration time to be overridden?
`imqJMSExpiration`	Long integer	`0`	Overriding value of expiration time, in milliseconds A value of `0` denotes an unlimited expiration time (message never expires).
`imqOverrideJMSPriority`	Boolean	`false`	Allow client-set priority level to be overridden?
`imqJMSPriority`	Integer	`4` (normal)	Overriding value of priority level (`0` to `9`)
`imqOverrideJMSHeadersToTemporaryDestinations`	Boolean	`false`	Apply overrides to temporary destinations?

Destination Attributes

Table 19–9 lists the attributes that can be set for a destination administered object.

Table 19–9   Destination Attributes

Attribute	Type	Default Value	Description
`imqDestinationName`	String	`Untitled_Destination_Object`	Name of physical destination The destination name may contain only alphanumeric characters (no spaces) and must begin with an alphabetic character or the underscore (_) or dollar sign ($) character. It may not begin with the characters `mq`.
`imqDestinationDescription`	String	None	Descriptive string for destination

20

JMS Resource Adapter Property Reference

This chapter describes the configuration properties of the Message Queue JMS Resource Adapter (JMS RA), which enables you to integrate Message Queue with any Java EE application server by means of the standard Java EE connector architecture (JCA). When plugged into an application server, the Resource Adapter allows applications deployed in that application server to use Message Queue to send and receive JMS messages.

The Message Queue JMS Resource Adapter exposes its configuration properties through three JavaBean components:

- The `ResourceAdapter` JavaBean (ResourceAdapter JavaBean) affects the behavior of the Resource Adapter as a whole.

- The `ManagedConnectionFactory` JavaBean (ManagedConnectionFactory JavaBean) affects connections created by the Resource Adapter for use by message-driven beans (MDBs).

- The `ActivationSpec` JavaBean (ActivationSpec JavaBean) affects message endpoints that represent MDBs in their interactions with the messaging system.

To set property values for these entities, you use the tools provided by your application server for configuration and deployment of the Resource Adapter and for deployment of MDBs.

This chapter lists and describes the configuration properties of the Message Queue JMS Resource Adapter. It contains the following sections:

- About Shared Topic Subscriptions for Clustered Containers

- ResourceAdapter JavaBean

- ManagedConnectionFactory JavaBean

- ActivationSpec JavaBean

About Shared Topic Subscriptions for Clustered Containers

The Message Queue JMS Resource Adapter provides a special feature called shared subscriptions for containers that support clustering, such as GlassFish Server. This feature enables clustered containers to share the load of processing messages for topic subscriptions across the instances of a cluster.

When this feature is enabled, the following behaviors apply:

- Attempts by multiple connections to use the same client id do not result in an exception, provided that the connections are from different instances in the cluster.

- Two or more subscriptions on the same topic with the same client id and (if the subscription is durable) the same durable subscription name are considered "shared"; that is, they are treated as a single subscription, with each message being sent to only one of the participating subscriptions.

The sharing of subscriptions relies on client id being set, not only for durable subscriptions (which always require client id) but for non-durable subscriptions (which do not normally require client id). If the subscription is being created by the resource adapter for use by a message-driven bean (MDB), and client id is not set, then the resource adapter will set the client id to the name of the MDB. However if the subscription is being created programmatically using the JMS API, and client id is not set, then an exception will be thrown.

Note that, in the EJB or web container, applications that create a connection using a connection factory are not permitted to set client id on the newly created connection, but must set it on the connection factory instead. This restriction is imposed by the EJB specification, though it applies to web components as well. There is no such restrictions in the application client container.

Disabling Shared Subscriptions

By default, the shared subscriptions feature is enabled. In some applications that use non-durable subscriptions, however, the shared behavior is not desired. In such cases, disable the shared subscriptions feature by setting the `useSharedSubscriptionInClusteredContainer` property to `false` on either the `ActivationSpec` or `ManagedConnectionFactory`, as appropriate:

- For an MDB, set the `ActivationSpec` property `useSharedSubscriptionInClusteredContainer` to `false`. Do this in exactly the same way as with other `ActivationSpec` properties, using annotations in the MDB itself or in the deployment descriptor `ejb-jar.xml` or `glassfish-ejb-jar.xml`. Alternatively, if the `glassfish-ejb-jar.xml` deployment descriptor specifies a connection factory using the `<mdb-connection-factory>` element, then the property can be configured on the connection factory instead, as described in the next item.

- For GlassFish applications creating a non-durable subscription using the JMS API rather than using an MDB, set the connection factory property `useSharedSubscriptionInClusteredContainer` to `false` using the GlassFish Administration Console, the GlassFish `asadmin` command or the resource descriptor `glassfish-resources.xml`.

Only set `useSharedSubscriptionInClusteredContainer` to `false` for non-durable subscriptions.

Consumer Flow Control When Shared Subscriptions Are Used

When shared subscriptions are being used, then consumer flow control operates slightly differently than is described in Client Runtime Message Flow Adjustments.

With a normal topic subscription, the maximum number of messages that can be held pending for any single subscriber, waiting to be consumed, is defined by the broker property `imq.autocreate.topic.consumerFlowLimit` for auto-created topics, or the destination property `consumerFlowLimit` for administratively-created topics. Both properties have a default value of 1000. This can be overridden on a per-connection basis by setting the connection factory property `imqConsumerFlowLimit` to a lower value than that defined for the topic.

When the subscription is shared, however, different logic applies. In this case, the limit is defined by the broker property `imq.autocreate.topic.sharedConsumerFlowLimit` for auto-created topics or the broker property `imq.admincreate.topic.sharedConsumerFlowLimit` for all administratively-created topics. It is not possible to set this limit on individual administratively-created topics. Both properties have a default value of 5. This can be overridden on a per-connection basis by setting the connection factory property `imqConsumerFlowLimit` to a lower value than that defined for the topic. Note that, as with all connection factory properties, this is specified using the `options` property of the managed connection factory.

ResourceAdapter JavaBean

The `ResourceAdapter` configuration configures the default JMS Resource Adapter behavior. Table 20–1 lists and describes the properties with which you can configure this JavaBean.

Table 20–1   Resource Adapter Properties

Property	Type	Default Value	Description
connectionURL[1]	String	`mq://localhost:7676/jms`	Message service address for connecting to the Message Queue service
brokerInstanceName	String	`imqbroker`	Name of broker instance
brokerPort	Integer	`7676`	Port number for connecting to broker
brokerBindAddress	String	Null	Address to which broker binds on host machine
			If null, the broker will bind to all addresses on the host machine.
userName[2]	String	`guest`	Default user name for connecting to Message Queue service
password[2]	String	`guest`	Default password for connecting to Message Queue service
addressListBehavior	String	`PRIORITY`	Order in which to attempt connection to Message Queue service:
			■ `PRIORITY`: Order specified in address list
			■ `RANDOM`: Random order
			Note: Reconnection attempts after a connection failure start with the broker whose connection failed and proceed sequentially through the address list, regardless of the value set for this property.
addressListIterations	Integer	`1`	Number of times to iterate through address list attempting to establish or reestablish a connection
reconnectEnabled	Boolean	`false`	Attempt to reestablish a lost connection?

Table 20–1   (Cont.) Resource Adapter Properties

Property	Type	Default Value	Description
reconnectAttempts	Integer	6	Number of times to attempt reconnection to each address in address list before moving on to next
reconnectInterval	Long integer	30000	Interval, in milliseconds, between reconnection attempts
brokerEnableHA	Boolean	false	Enable high availability?
clusterID	String	None	Cluster identifier
			If specified, only brokers with the same cluster identifier can be clustered together. In the event of broker failure, client connections will fail over only to brokers with the same cluster identifier as the original broker. If not specified, client connections can fail over to any other broker with an unspecified cluster identifier.
			For standalone brokers (those not belonging to a cluster), this property is ignored.
			The identifier may contain only alphabetic letters (A-Z, a-z), numeric digits (0-9), and the underscore character (_).
brokerID	String	None	Broker identifier
			For brokers using a JDBC-based persistent data store, this string is appended to the names of all database tables to make them unique in the case where more than one broker instance is using the same database. For brokers using a file-based data store, this property is ignored.
			In an enhanced cluster, each broker must have a unique broker identifier.
			The identifier may contain only alphabetic letters (A-Z, a-z), numeric digits (0-9), and the underscore character (_).

[1] Must be specified
[2] Required

ManagedConnectionFactory JavaBean

A managed connection factory is used to create connections managed by the resource adapter. Table 20–2 shows the properties of the `ManagedConnectionFactory` JavaBean; if set, these properties override the corresponding properties of the `ResourceAdapter` JavaBean.

Table 20–2   Managed Connection Factory Properties

Property	Type	Default Value	Description
addressList	String	Value of connectionURL property of ResourceAdapter JavaBean (see Table 20–1)	List of message service addresses for connecting to Message Queue service
userName[1]	String	guest	User name for connecting to Message Queue service
password[1]	String	guest	Password for connecting to Message Queue service
clientID	String	None	Client identifier for connections to Message Queue service
addressListBehavior	String	PRIORITY	Order in which to attempt connection to Message Queue service: ■ PRIORITY: Order specified in address list ■ RANDOM: Random order **Note:** Reconnection attempts after a connection failure start with the broker whose connection failed and proceed sequentially through the address list, regardless of the value set for this property.
addressListIterations	Integer	1	Number of times to iterate through address list attempting to establish or reestablish a connection
reconnectEnabled	Boolean	false	Attempt to reestablish a lost connection?
reconnectAttempts	Integer	6	Number of times to attempt reconnection to each address in address list before moving on to next

Table 20–2   (Cont.) Managed Connection Factory Properties

Property	Type	Default Value	Description
reconnectInterval	Long integer	30000	Interval, in milliseconds, between reconnection attempts
options[1]	String	None	A list of additional connection factory properties to be used when creating connections to a Message Queue broker.
			When specified, the value of options must be a comma-separated list of connection factory properties and their values, in the form:
			propertyName=value
			If *value* contains a comma or an equals sign, precede the symbol with a backslash (\) or enclose the entire value in quotes; for example:
			`prop1=comma\,val,prop2="equals=val"`
			The options property cannot specify properties that are configured internally or that have their own setter methods, specifically: imqReconnectEnabled, imqReconnectAttempts, imqReconnectInterval, imqDefaultUsername, imqDefaultPassword, imqAddressList, imqAddressListIterations. Any values specified in options for these properties are ignored.
useSharedSubscriptionInClusteredContainer	Boolean	true	Controls whether topic subscriptions created using this ManagedConnectionFactory will be shared when running in a clustered container, as described in About Shared Topic Subscriptions for Clustered Containers.
			Set to true (the default) to share subscriptions. The clientID property must also be set, even if the subscription is nondurable.
			Set to false to not share subscriptions. This setting should only be used for nondurable subscriptions. The clientID property does not need to be set.

---

[1]   Optional

ActivationSpec JavaBean

Table 20–3 shows the configurable properties of the `ActivationSpec` JavaBean. These properties are used by the application server when instructing the Resource Adapter to activate a message endpoint and associate it with a message-driven bean.

Table 20–3   ActivationSpec Properties

Property	Type	Default Value	Description
addressList[1,2]	String	Value of `connectionURL` property of `ResourceAdapter` JavaBean (see Table 20–1)	Message service address for connecting to Message Queue service
userName[1,2]	String	Inherited from `ResourceAdapter` JavaBean (see Table 20–1)	User name for connecting to Message Queue service.
password[1,2]	String	Inherited from `ResourceAdapter` JavaBean (see Table 20–1)	Password for connecting to Message Queue service.
clientId[3]	String	None	Client ID for connections to Message Queue service This property must be set if `subscriptionDurability` is set to `Durable`.
addressListBehavior[1,2]	String	Inherited from `ResourceAdapter` JavaBean (see Table 20–1)	Order in which to attempt connection or reconnection to Message Queue service: PRIORITY: order specified in address list RANDOM: Random order
addressListIterations[1,2]	Integer	Inherited from `ResourceAdapter` JavaBean (see Table 20–1)	Number of times to iterate through addressList attempting to establish or reestablish a connection. When this limit is reached an exception will be thrown and a new connection attempt will begin, with no limit.
reconnectAttempts[1,2]	Integer	Inherited from `ResourceAdapter` JavaBean (see Table 20–1)	Number of times to attempt connection or reconnection to each address in addressList before moving on to next.
reconnectInterval1[1,2]	Integer	Inherited from `ResourceAdapter` JavaBean (see Table 20–1)	Interval, in milliseconds, between reconnection attempts
destination[3]	String	None	Name of destination from which to consume messages The value must be that of the `destinationName` property for a Message Queue destination administered object.

Table 20–3   (Cont.) ActivationSpec Properties

Property	Type	Default Value	Description
destinationType[3]	String	None	Type of destination specified by `destination` property: ■ `javax.jms.Queue`: Queue destination ■ `javax.jms.Topic`: Topic destination
messageSelector[1,3]	String	None	Message selector for filtering messages delivered to consumer
subscriptionName[3]	String	None	Name for durable subscriptions This property must be set if `subscriptionDurability` is set to `Durable`.
subscriptionDurability[3]	String	NonDurable	Durability of consumer for topic destination: ■ `Durable`: Durable consumer ■ `NonDurable`: Nondurable consumer This property is valid only if `destinationType` is set to `javax.jms.Topic`, and is optional for nondurable subscriptions and required for durable ones. If set to `Durable`, the `clientID` and `subscriptionName` properties must also be set.
acknowledgeMode[1,3]	String	Auto-acknowledge	Acknowledgment mode: ■ `Auto-acknowledge`: Auto-acknowledge mode ■ `Dups-ok-acknowledge`: Dups-OK-acknowledge mode
customAcknowledgeMode	String	None	Acknowledgment mode for MDB message consumption Valid values are `No_acknowledge` or null. You can use no-acknowledge mode only for a nontransacted, nondurable topic subscription; if you use this setting with a transacted subscription or a durable subscription, subscription activation will fail.
endpointExceptionRedeliveryAttempts	Integer	6	Number of times to redeliver a message when MDB throws an exception during message delivery
endpointExceptionRedeliveryInterval	integer	500	The amount of time, in milliseconds, to wait before the next redelivery attempt.

Table 20–3   (Cont.) ActivationSpec Properties

Property	Type	Default Value	Description
sendUndeliverableMsgsToDMQ	Boolean	true	Place message in dead message queue when MDB throws a runtime exception and number of redelivery attempts exceeds the value of endpointExceptionRedeliveryAttempts?
			If false, the Message Queue broker will attempt redelivery of the message to any valid consumer, including the same MDB.
options[1]	String	None	A list of additional connection factory properties to be used when creating connections to a Message Queue broker.
			When specified, the value of options must be a comma-separated list of connection factory properties and their values, in the form:
			propertyName=value
			If *value* contains a comma or an equals sign, precede the symbol with a backslash (\) or enclose the entire value in quotes; for example:
			prop1=comma\,val,prop2="equals=val"
			The options property cannot be used to specify properties that are configured internally or that have their own setter methods, specifically: imqReconnectEnabled, imqReconnectAttempts, imqReconnectInterval, imqDefaultUsername, imqDefaultPassword, imqAddressList, imqAddressListIterations, imqAddressListBehavior. Any values specified in options for these properties are ignored.

Table 20–3   (Cont.) ActivationSpec Properties

Property	Type	Default Value	Description
useSharedSubscriptionInClusteredContainer	Boolean	true	Controls whether topic subscriptions created using this ActivationSpec will be shared when running in a clustered container, as described in About Shared Topic Subscriptions for Clustered Containers.
			Set to true (the default) to share subscriptions.
			Set to false to not share subscriptions. This setting should only be used for nondurable subscriptions.

[1] Optional

[2] Property specific to Message Queue JMS Resource Adapter

[3] Standard Enterprise JavaBean (EJB) and Java EE Connector Architecture (CA) property

Note that there is no reconnectEnabled property for the ActivationSpec JavaBean.

21

Metrics Information Reference

This chapter describes the metrics information that a Message Queue broker can provide for monitoring, tuning, and diagnostic purposes. This information can be made available in a variety of ways:

- In a log file (see Sending Metrics Data to Log Files)

- Interactively with the Command utility's `imqcmd metrics` subcommand (see Using the Command Utility)

- In metrics messages sent to a metrics topic destination (see Using the Message-Based Monitoring API)

- Through JMX MBeans that can be accessed programmatically by Java applications using the JMX Administration API.

The tables in this chapter list the kinds of metrics information available and the forms in which it can be provided. For metrics provided through the Command utility's `imqcmd metrics` subcommand, the tables list the metric type with which they can be requested; for those provided in metrics messages, the tables list the metrics topic destination to which they are delivered. All the metrics information in this chapter can be accessed progamatically using the JMX Administration API as described in the *Open Message Queue Developer's Guide for JMX Clients*

The chapter consists of the following sections:

- JVM Metrics

- Brokerwide Metrics

- Connection Service Metrics

- Physical Destination Metrics

JVM Metrics

Table 21–1 shows the metrics information that the broker reports for the broker process JVM (Java Virtual Machine) heap.

Table 21–1   JVM Metrics

Metrics Quantity	Description	Log File?	metrics bkrMetric Type	Metrics Topic
JVM heap: total memory	Current total memory, in bytes	Yes	cxn	mq.metrics.jvm
JVM heap: free memory	Amount of memory currently available for use, in bytes	Yes	cxn	mq.metrics.jvm
JVM heap: max memory	Maximum allowable heap size, in bytes	Yes	None	mq.metrics.jvm

Brokerwide Metrics

Table 21–2 shows the brokerwide metrics information that the broker reports.

Table 21–2   Brokerwide Metrics

Metrics Quantity	Description	Log File?	metrics bkrMetric Type	Metrics Topic
Connections				
Num connections	Total current number of connections for all connection services	Yes	cxn	mq.metrics.broker
Num threads	Total current number of threads for all connection services	Yes	cxn	None
Min threads	Total minimum number of threads for all connection services	Yes	cxn	None
Max threads	Total maximum number of threads for all connection services	Yes	cxn	None
Stored Messages				
Num messages	Current number of payload messages stored in memory and persistent store	No	None[1]	mq.metrics.broker
Total message bytes	Total size in bytes of payload messages currently stored in memory and persistent store	No	None[1]	mq.metrics.broker
Message Flow				
Num messages in	Cumulative number of payload messages received since broker started	Yes	ttl	mq.metrics.broker
Num messages out	Cumulative number of payload messages sent since broker started	Yes	ttl	mq.metrics.broker
Rate messages in	Current rate of flow of payload messages into broker	Yes	rts	None
Rate messages out	Current rate of flow of payload messages out of broker	Yes	rts	None
Message bytes in	Cumulative size in bytes of payload messages received since broker started	Yes	ttl	mq.metrics.broker
Message bytes out	Cumulative size in bytes of payload messages sent since broker started	Yes	ttl	mq.metrics.broker
Rate message bytes in	Current rate of flow of payload message bytes into broker	Yes	rts	None

Table 21–2   (Cont.) Brokerwide Metrics

Metrics Quantity	Description	Log File?	`metrics bkrMetric Type`	Metrics Topic
Rate message bytes out	Current rate of flow of payload message bytes out of broker	Yes	`rts`	None
Num packets in	Cumulative number of payload and control packets received since broker started	Yes	`ttl`	`mq.metrics.broker`
Num packets out	Cumulative number of payload and control packets sent since broker started	Yes	`ttl`	`mq.metrics.broker`
Rate packets in	Current rate of flow of payload and control packets into broker	Yes	`rts`	None
Rate packets out	Current rate of flow of payload and control packets out of broker	Yes	`rts`	None
Packet bytes in	Cumulative size in bytes of payload and control packets received since broker started	Yes	`ttl`	`mq.metrics.broker`
Packet bytes out	Cumulative size in bytes of payload and control packets sent since broker started	Yes	`ttl`	`mq.metrics.broker`
Rate packet bytes in	Current rate of flow of payload and control packet bytes into broker	Yes	`rts`	None
Rate packet bytes out	Current rate of flow of payload and control packet bytes out of broker	Yes	`rts`	None
Destinations				
Num destinations	Current number of physical destinations	No	None	`mq.metrics.broker`

[1]   Use `query bkr` command instead

Connection Service Metrics

Table 21–3 shows the metrics information that the broker reports for individual connection services.

Table 21–3   Connection Service Metrics

Metrics Quantity	Description	Log File?	`metrics svcMetric Type`	Metrics Topic
Connections				
Num connections	Current number of connections	No	`cxn`[1]	None
Num threads	Current number of threads	No	`cxn`[1]	None
Min threads	Minimum number of threads assigned to service	No	`cxn`	None
Max threads	Maximum number of threads assigned to service	No	`cxn`	None
Message Flow				
Num messages in	Cumulative number of payload messages received through connection service since broker started	No	`ttl`	None

Table 21–3 (Cont.) Connection Service Metrics

Metrics Quantity	Description	Log File?	`metrics svcMetric` Type	Metrics Topic
Num messages out	Cumulative number of payload messages sent through connection service since broker started	No	`ttl`	None
Rate messages in	Current rate of flow of payload messages into broker through connection service	No	`rts`	None
Rate messages out	Current rate of flow of payload messages out of broker through connection service	No	`rts`	None
Message bytes in	Cumulative size in bytes of payload messages received through connection service since broker started	No	`ttl`	None
Message bytes out	Cumulative size in bytes of payload messages sent through connection service since broker started	No	`ttl`	None
Rate message bytes in	Current rate of flow of payload message bytes into broker through connection service	No	`rts`	None
Rate message bytes out	Current rate of flow of payload message bytes out of broker through connection service	No	`rts`	None
Num packets in	Cumulative number of payload and control packets received through connection service since broker started	No	`ttl`	None
Num packets out	Cumulative number of payload and control packets sent through connection service since broker started	No	`ttl`	None
Rate packets in	Current rate of flow of payload and control packets into broker through connection service	No	`rts`	None
Rate packets out	Current rate of flow of payload and control packets out of broker through connection service	No	`rts`	None
Packet bytes in	Cumulative size in bytes of payload and control packets received through connection service since broker started	No	`ttl`	None
Packet bytes out	Cumulative size in bytes of payload and control packets sent through connection service since broker started	No	`ttl`	None
Rate packet bytes in	Current rate of flow of payload and control packet bytes into broker through connection service	No	`rts`	None
Rate packet bytes out	Current rate of flow of payload and control packet bytes out of broker through connection service	No	`rts`	None

[1]   Also available with `query svc` command

Physical Destination Metrics

Table 21–4 shows the metrics information that the broker reports for individual destinations.

Table 21–4   Physical Destination Metrics

Metrics Quantity	Description	Log File?	metrics dstMetric Type	Metrics Topic
Message Consumers				
Num consumers	Current number of associated message consumers For queue destinations, this attribute includes both active and backup consumers. For topic destinations, it includes both nondurable and (active and inactive) durable subscribers and is equivalent to "Num active consumers."	No	con	`mq.metrics.destination.queue.`*queueName*`mq.metrics.destination.topic.`*topicName*
Peak num consumers	Peak number of associated message consumers since broker started For queue destinations, this attribute includes both active and backup consumers. For topic destinations, it includes both nondurable and (active and inactive) durable subscribers and is equivalent to "Peak num active consumers."	No	con	`mq.metrics.destination.queue.`*queueName*`mq.metrics.destination.topic.`*topicName*
Avg num consumers	Average number of associated message consumers since broker started For queue destinations, this attribute includes both active and backup consumers. For topic destinations, it includes both nondurable and (active and inactive) durable subscribers and is equivalent to "Avg num active consumers."	No	con	`mq.metrics.destination.queue.`*queueName*`mq.metrics.destination.topic.`*topicName*

Table 21–4   (Cont.) Physical Destination Metrics

Metrics Quantity	Description	Log File?	metrics dstMetric Type	Metrics Topic
Num active consumers	Current number of associated active message consumers For topic destinations, this attribute includes both nondurable and (active and inactive) durable subscribers and is equivalent to "Num consumers."	No	con	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Peak num active consumers	Peak number of associated active message consumers since broker started For topic destinations, this attribute includes both nondurable and (active and inactive) durable subscribers and is equivalent to "Peak num consumers."	No	con	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Avg num active consumers	Average number of associated active message consumers since broker started For topic destinations, this attribute includes both nondurable and (active and inactive) durable subscribers and is equivalent to "Avg num consumers."	No	con	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Num backup consumers[1]	Current number of associated backup message consumers	No	con	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Peak num backup consumers[1]	Peak number of associated backup message consumers since broker started	No	con	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Avg num backup consumers[1]	Average number of associated backup message consumers since broker started	No	con	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Stored Messages				
Num messages	Current number of messages stored in memory and persistent store	No	conttlrts[2]	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*

Table 21–4   (Cont.) Physical Destination Metrics

Metrics Quantity	Description	Log File?	metrics dstMetric Type	Metrics Topic
Num messages remote	Current number of messages stored in memory and persistent store that were sent from a remote broker in a cluster. This number does not include messages included in transactions.	No	Not Available[3]	Not Available
Peak num messages	Peak number of messages stored in memory and persistent store since broker started	No	conttlrts	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Avg num messages	Average number of messages stored in memory and persistent store since broker started	No	conttlrts	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Total message bytes	Current total size in bytes of messages stored in memory and persistent store	No	ttlrts[2]	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Total message bytes remote	Current total size in bytes of messages stored in memory and persistent store that were sent from a remote broker in a cluster. This value does not include messages included in transactions.	No	Not Available[3]	Not Available
Peak total message bytes	Peak total size in bytes of messages stored in memory and persistent store since broker started	No	ttlrts	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Avg total message bytes	Average total size in bytes of messages stored in memory and persistent store since broker started	No	ttlrts	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Message Flow				
Num messages in	Cumulative number of messages received since broker started	No	ttl	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*
Num messages out	Cumulative number of messages sent since broker started	No	ttl	mq.metrics.destination.queue.*queueName*mq.metrics.destination.topic.*topicName*

Table 21–4   (Cont.) Physical Destination Metrics

Metrics Quantity	Description	Log File?	`metrics dst`Metric Type	Metrics Topic
Msg bytes in	Cumulative size in bytes of messages received since broker started	No	`ttl`	`mq.metrics.destination.queue.`*queueName*`mq.metrics.destination.topic.`*topicName*
Msg bytes out	Cumulative size in bytes of messages sent since broker started	No	`ttl`	`mq.metrics.destination.queue.`*queueName*`mq.metrics.destination.topic.`*topicName*
Peak message bytes	Size in bytes of largest single message received since broker started	No	`ttlrts`	`mq.metrics.destination.queue.`*queueName*`mq.metrics.destination.topic.`*topicName*
Rate num messages in	Current rate of flow of messages received	No	`rts`	None
Rate num messages out	Current rate of flow of messages sent	No	`rts`	None
Rate msg bytes in	Current rate of flow of message bytes received	No	`rts`	None
Rate msg bytes out	Current rate of flow of message bytes sent	No	`rts`	None
Disk Utilization				
Disk reserved[4]	Amount of disk space, in bytes, reserved for destination	No	`dsk`	`mq.metrics.destination.queue.`*queueName*`mq.metrics.destination.topic.`*topicName*
Disk used[4]	Amount of disk space, in bytes, currently in use by destination	No	`dsk`	`mq.metrics.destination.queue.`*queueName*`mq.metrics.destination.topic.`*topicName*
Disk utilization ratio[4]	Ratio of disk space in use to disk space reserved for destination	No	`dsk`	`mq.metrics.destination.queue.`*queueName*`mq.metrics.destination.topic.`*topicName*

[1]   Queue destinations only

[2]   Also available with `query dst` command

[3]   Available only with `imqcmd query dst` command

[4]   File-based persistence only

22

JES Monitoring Framework Reference

This chapter describes the monitoring information items that Message Queue exposes through the Sun Java Enterprise System Monitoring Framework (JESMF), using the Monitoring Framework's Common Monitoring Model (CMM). It contains the following sections:

- Common Attributes
- Message Queue Product Information
- Broker Information
- Port Mapper Information
- Connection Service Information
- Destination Information
- Persistent Store Information
- User Repository Information

Common Attributes

The attributes listed in Table 22–1 are common to all (or almost all) CMM objects.

Table 22–1   JESMF Common Object Attributes

Attribute	Description
Name	Object name
Caption	Short description
Description	Full description
LastUpdateTime	Time last updated
OperationalStatus	Current status (for example, OK or DORMANT)
StatusDescriptions	Description of status
OperationalStatusLastChange	Time of last change in operational status

Message Queue Product Information

Table 22–2 shows attributes of the Message Queue product itself that can be accessed with JESMF.

Table 22–2   JESMF-Accessible Message Queue Product Attributes

Attribute	Description
ProductName	Product name
ProductIdentifyingNumber	Identifying number of product, in the form
	■ $urn{:}uuid{:}xxxxxxxx\text{-}xxxx\text{-}xxxx\text{-}xxxx\text{-}xxxxxxxxxxxx$
	Value changes for every version.
ProductVendor	Vendor name
ProductVersion	Version number
RevisionNumber	Revision number
BuildNumber	Build number
PatchID	Patch identifier (if any)
CollectionID	Identification key for installed product object
	Differentiates among product installations; usually identifies the installation location.
InstallDate	Installation date

Broker Information

Table 22–3 shows the JESMF -accessible attributes pertaining to each broker instance.

Table 22–3   JESMF-Accessible Message Queue Broker Attributes

Attribute	Description
PrimaryOwnerName	Name of primary system owner (broker property imq.primaryowner.name; see Table 17–13)
PrimaryOwnerContact	Contact information for primary system owner (broker property imq.primaryowner.contact; see Table 17–13)
Roles	Array of strings denoting broker's roles (taken from broker properties imq.broker.adminDefinedRoles.name*n*; see Table 17–13)
StartupTime	Time of last startup (date and time in milliseconds)
URL	URL of Port Mapper
ConfigurationDirectory	Broker instance directory (for example, /var/imq/instances/mybroker)
DirectoryName	Distinguished name of directory (for example, LDAP) entry where static information about application is stored
	An empty string indicates that no information about the application is available in the directory.

Port Mapper Information

The attributes shown in Table 22–4 provide information about a broker's Port Mapper.

Table 22–4   JESMF-Accessible Message Queue Port Mapper Attributes

Attribute	Description
LabeledURI	URI for accessing Port Mapper, in the form
	■ mq://*hostName*:*portNumber*

Table 22–4   (Cont.) JESMF-Accessible Message Queue Port Mapper Attributes

Attribute	Description
Secured	Is Port Mapper access secure (SSL/TLS)?

Connection Service Information

Table 22–5 shows the JESMF -accessible attributes pertaining to each connection service.

Table 22–5   JESMF-Accessible Message Queue Connection Service Attributes

Attribute	Description
LabeledURI	URI for accessing connection service, in the form ■ `mq://`*hostName*`:`*portNumber*`/`*serviceName* if dynamically allocated, or ■ `mqtcp://`*hostName*`:`*servicePort*`/`*serviceName* or ■ `mqssl://`*hostName*`:`*servicePort*`/`*serviceName* if statically assigned
Secured	Is connection service access secure (SSL/TLS)?
ConnectionsCount	Current number of connections
NumConnectionsCreated	Cumulative number of connections created since broker started
FailedConnectionsCount	Cumulative number of connections rejected since broker started
CurrentNumberOfThreads	Current number of threads actively handling connections
MinThreadPoolSize	Minimum number of threads maintained in connection service's thread pool (broker property `imq.`*serviceName*`.min_threads`; see Table 17–1)
MaxThreadPoolSize	Number of threads beyond which no new threads are added to thread pool for use by connection service (broker property `imq.`*serviceName*`.max_threads`; see Table 17–1)
NumProducers	Current number of message producers
NumConsumers	Current number of message consumers
NumMsgsIn	Cumulative number of messages received since broker started
NumMsgsOut	Cumulative number of messages sent since broker started
InBytesCount	Cumulative size in bytes of messages received since broker started
OutBytesCount	Cumulative size in bytes of messages sent since broker started
NumPktsIn	Cumulative number of packets received since broker started
NumPktsOut	Cumulative number of packets sent since broker started
PktBytesIn	Cumulative size in bytes of packets received since broker started
PktBytesOut	Cumulative size in bytes of packets sent since broker started

Destination Information

Table 22–6 shows the JESMF -accessible attributes pertaining to each destination. Each of these attributes corresponds to a Message Queue physical destination property; see Table 18–1 for further information.

Table 22–6   JESMF-Accessible Message Queue Destination Attributes

Attribute	Corresponding Property	Description
Type		Destination type (q = queue, t = topic)
MaxNumMsgs	maxNumMsgs	Maximum number of unconsumed messages
MaxBytesPerMsg	maxBytesPerMsg	Maximum size, in bytes, of any single message
MaxTotalMsgBytes	maxTotalMsgBytes	Maximum total memory, in bytes, for unconsumed messages
LimitBehavior	limitBehavior	Broker behavior when memory-limit threshold reached
MaxNumProducers[1]	maxNumProducers	Maximum number of associated message producers
MaxNumActiveConsumers[2]	maxNumActiveConsumers	Maximum number of associated active message consumers in load-balanced delivery
MaxNumBackupConsumers[2]	maxNumBackupConsumers	Maximum number of associated backup message consumers in load-balanced delivery
ConsumerFlowLimit	consumerFlowLimit	Maximum number of messages delivered to consumer in a single batch
LocalOnly[1]	isLocalOnly	Local delivery only?
LocalDeliveryPreferred[1,2]	localDeliveryPreferred	Local delivery preferred?
UseDMQ[1]	useDMQ	Send dead messages to dead message queue?

[1] Does not apply to dead message queue
[2] Queue destinations only

Persistent Store Information

The attributes shown in Table 22–7 pertain to the persistent data store.

Table 22–7   JESMF-Accessible Message Queue Persistent Store Attributes

Attribute	Description
AccessInfo	URL for accessing JDBC database
InfoFormat	Format of AccessInfo attribute (URL)
JDBCDriver	JDBC driver
UserName	User name for authentication

User Repository Information

The attributes shown in Table 22–8 pertain to the LDAP user repository.

Table 22–8   JESMF-Accessible Message Queue User Repository Attributes

Attribute	Description
AccessInfo	URL for accessing LDAP server
InfoFormat	Format of AccessInfo attribute (URL)
Base	Root or base node for user lookup
GroupBase	Root or base node for group lookup
UserName	User name for authentication

Part IV

Appendixes

A

Distribution-Specific Locations of Message Queue Data

Message Queue data is stored in different locations based on the distribution used to install Message Queue. The tables that follow show the location of various types of Message Queue data for the following types of installations:

- Installations from an IPS image
- Installations of Previous Message Queue Versions from Solaris SVR4 Packages
- Installations of Previous Message Queue Versions from Linux RPMs

In the tables, *instanceName* denotes the name of the broker instance with which the data is associated.

Installations from an IPS image

Table A–1 shows the location of Message Queue data when Message Queue is installed from an IPS image, such as when the Message Queue installer is used. Locations denote the `IMQ_HOME` and `IMQ_VARHOME` directory variables defined in Directory Variable Conventions.

Table A–1   Message Queue Data Locations for Installations from an IPS Image

Data Category	Location
Command line executable files	`IMQ_HOME/bin`
Broker instance configuration properties	`IMQ_VARHOME/instances/`*instanceName*`/props/config.properties`
Broker configuration file templates	`IMQ_HOME/lib/props/broker/`
Persistent data store (messages, destinations, durable subscriptions, transactions, acknowledgements)	`IMQ_VARHOME/instances/`*instanceName*`/fs370/`or a JDBC-accessible data store
Broker instance log file directory (default location)	`IMQ_VARHOME/instances/`*instanceName*`/log/`
Administered objects (object store)	Local directory of your choice or an LDAP server

Table A–1   (Cont.) Message Queue Data Locations for Installations from an IPS Image

Data Category	Location
Security: user repository	`IMQ_VARHOME/instances/`*instanceName*`/etc/passwd`or an LDAP server
Security: access control file (default location)	`IMQ_ VARHOME/instances/`*instanceName*`/etc/accesscontrol.properties`
Security: password file directory (default location)	`IMQ_HOME/etc/`
Security: example password file	`IMQ_HOME/etc/passfile.sample`
Security: broker's key store file location	`IMQ_HOME/etc/`
JavaDoc API documentation	`IMQ_HOME/javadoc/index.html`
Example applications and configurations	`IMQ_HOME/examples/`
Java archive (`.jar`), Web archive (`.war`), and Resource Adapter archive (`.rar`) files	`IMQ_HOME/lib/`
External resource (`.jar`) files such as JDBC drivers, JAAS login modules, and so forth	`IMQ_HOME/lib/ext`
JMS Bridge DTD file	`IMQ_HOME/lib/dtd`

Installations of Previous Message Queue Versions from Solaris SVR4 Packages

> **Note:**   Message Queue is no longer distributed as Solaris SVR4 packages. The following information is provided as a point of reference when upgrading from previous versions of Message Queue that were distributed as Solaris SVR4 packages.

Table A–2 shows the location of Message Queue data when Message Queue was installed from Solaris SVR4 packages.

Table A–2   Message Queue Data Locations for Installations from Solaris SVR4 Packages

Data Category	Location
Command line executable files	`/usr/bin`
Broker instance configuration properties	`/var/imq/instances/`*instanceName*`/props/config.properties`
Broker configuration file templates	`/usr/share/lib/imq/props/broker/`

Table A–2   (Cont.) Message Queue Data Locations for Installations from Solaris SVR4

Data Category	Location
Persistent data store (messages, destinations, durable subscriptions, transactions, acknowledgements)	`/var/imq/instances/`*instanceName*`/fs370`or a JDBC-accessible data store
Broker instance log file directory (default location)	`/var/imq/instances/`*instanceName*`/log/`
Administered objects (object store)	Local directory of your choice or an LDAP server
Security: user repository	`/var/imq/instances/`*instanceName*`/etc/passwd`or an LDAP server
Security: access control file (default location)	`/var/imq/instances/`*instanceName*`/etc/accesscontrol.properties`
Security: password file directory (default location)	`/var/imq/instances/`*instanceName*`/etc/`
Security: example password file	`/etc/imq/passfile.sample`
Security: broker's key store file location	`/etc/imq/`
JavaDoc API documentation	`/usr/share/javadoc/imq/index.html`
Example applications and configurations	`/usr/demo/imq/`
Java archive (`.jar`), Web archive (`.war`), and Resource Adapter archive (`.rar`) files	`/usr/share/lib/imq`
External resource (`.jar`) files such as JDBC drivers, JAAS login modules, and so forth	`/usr/share/lib/imq/ext`
JMS Bridge DTD file	`/usr/share/lib/imq/dtd`

Installations of Previous Message Queue Versions from Linux RPMs

> **Note:** Message Queue is no longer distributed as Linux RPMs. The following information is provided as a point of reference when upgrading from previous versions of Message Queue that were distributed as Linux RPMs.

Table A–3 shows the location of Message Queue data when Message Queue was installed from Linux RPMs.

Table A–3   Message Queue Data Locations for Installations from Linux RPMs

Data Category	Location
Command line executable files	`/opt/sun/mq/bin`
Broker instance configuration properties	`/var/opt/sun/mq/instances/`*instanceName*`/props/config.propertie` `s`
Broker configuration file templates	`/opt/sun/mq/private/share/lib/props/`
Persistent data store (messages, destinations, durable subscriptions, transactions, acknowledgements)	`/var/opt/sun/mq/instances/`*instanceName*`/fs370/`or a JDBC-accessible data store
Broker instance log file directory (default location)	`/var/opt/sun/mq/instances/`*instanceName*`/log/`
Administered objects (object store)	Local directory of your choice or an LDAP server
Security: user repository	`/var/opt/sun/mq/instances/`*instanceName*`/etc/passwd`or an LDAP server
Security: access control file (default location)	`/var/opt/sun/mq/instances/`*instanceName*`/etc/accesscontrol.prop` `erties`
Security: password file directory (default location)	`/var/opt/sun/mq/instances/`*instanceName*`/etc/`
Security: example password file	`/etc/opt/sun/mq/passfile.sample`
Security: broker's key store file location	`/etc/opt/sun/mq/`
JavaDoc API documentation	`/opt/sun/mq/javadoc/index.html`
Example applications and configurations	`/opt/sun/mq/examples/`
Java archive (`.jar`), Web archive (`.war`), and Resource Adapter archive (`.rar`) files	`/opt/sun/mq/share/lib/`
External resource (`.jar`) files such as JDBC drivers, JAAS login modules, and so forth	`/opt/sun/mq/share/lib/ext`
Shared library (`.so`) files	`/opt/sun/mq/lib/`
JMS Bridge DTD file	`/opt/sun/mq/share/lib/dtd`

B

Stability of Message Queue Interfaces

Message Queue uses many interfaces that can help administrators automate tasks. This appendix classifies the interfaces according to their stability. The more stable an interface is, the less likely it is to change in subsequent versions of the product.

Any interface that is not listed in this appendix is private and not for customer use.

Classification Scheme

Table B–1 describes the stability classification scheme.

Table B–1 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 (for example, 3.0 or 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 that customers not create explicit dependencies on unstable interfaces.

Interface Stability

Table B–2 lists the interfaces and their classifications.

Table B–2 Stability of Message Queue Interfaces

Interface	Classification
Command Line Interfaces	
`imqbrokerd` command line interface	Evolving
`imqadmin` command line interface	Unstable

Table B–2   (Cont.) Stability of Message Queue Interfaces

Interface	Classification
`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
`imqbridgemgr` command line interface	Evolving
Output from `imqbrokerd`, `imqadmin`, `imqcmd`, `imqdbmgr`, `imqkeytool`, `imqobjmgr`, `imqusermgr`	Unstable
Commands	
`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
`imqbridgemgr` command	Evolving
APIs	
JMS API (`javax.jms`)	Standard
JAXM API (`javax.xml`)	Standard
C-API	Evolving
C-API environment variables	Unstable
Message-based monitoring API	Evolving
Administered Object API (`com.sun.messaging`)	Evolving
.jar and .war Files	
`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
`imqjmsra.rar` location and name	Evolving
`imqxm.jar` location and name	Evolving
`jaxm-api.jar` location and name	Evolving

Table B–2   (Cont.) Stability of Message Queue Interfaces

Interface	Classification
`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
Files	
Broker log file location and content format	Unstable
password file	Unstable
`accesscontrol.properties` file	Unstable
System Destinations	
`mq.sys.dmq` destination	Stable
`mq.metrics.*` destinations	Evolving
Configuration Properties	
Message Queue JMS Resource Adapter configuration properties	Evolving
Message Queue JMS Resource Adapter JavaBean and ActivationSpec configuration properties	Evolving
Message Properties and Formats	
Dead message queue message properties, `JMS_SUN_*`	Evolving
Message Queue client message properties: `JMS_SUN_*`	Evolving
JMS message format for metrics or monitoring messages	Evolving
Miscellaneous	
Message Queue JMS Resource Adapter package, `com.sun.messaging.jms.ra`	Evolving
JDBC schema for storage of persistent messages	Evolving

C

HTTP/HTTPS Support

Message Queue includes support for Java clients to communicate with a message broker by means of the HTTP or secure HTTP (HTTPS) transport protocols, rather than through a direct TCP connection. (HTTP/ HTTPS support is not available for C clients.) Because HTTP/ HTTPS connections are normally allowed through firewalls, this allows client applications to be separated from the broker by a firewall.
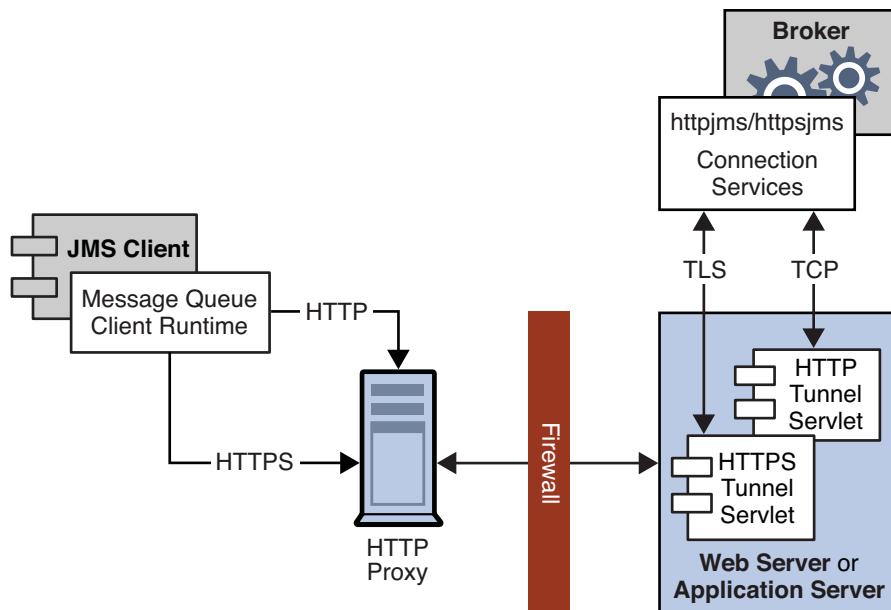
This appendix describes the architecture used to enable HTTP/ HTTPS support and explains the setup work needed to allow Message Queue clients to use such connections. It has the following sections:

- HTTP/HTTPS Support Architecture
- Enabling HTTP/HTTPS Support
- Troubleshooting

HTTP/HTTPS Support Architecture

Message Queue's support architecture is very similar for both HTTP and HTTPS support, as shown in Figure C–1:

- On the client side, an HTTP or HTTPS transport driver (part of the Message Queue client runtime) encapsulates each message into an HTTP request and makes sure that these requests are transmitted in the correct sequence.

- If necessary, the client can use an HTTP proxy server to communicate with the broker. The proxy's address is specified using command line options when starting the client; seeUsing an HTTP Proxy for more information.

- An HTTP or HTTPS tunnel servlet (both bundled with Message Queue) is loaded into an application server or Web server on the broker side and used to pull payload messages from client HTTP requests before forwarding them to the broker. The tunnel servlet also sends broker messages back to the client in response to the client's HTTP requests. A single tunnel servlet can be used to access multiple brokers.

- On the broker side, the `httpjms` or `httpsjms` connection service unwraps and demultiplexes incoming messages from the corresponding tunnel servlet.

Figure C–1   HTTP/HTTPS Support Architecture

The main difference between HTTP and HTTPS connections is that in the HTTPS case (`httpsjms` connection service), the tunnel servlet has a secure connection to both the client application and the broker. The secure connection to the broker is established by means of the Secure Socket Layer (SSL) protocol. Message Queue's SSL-enabled HTTPS tunnel servlet passes a self-signed certificate to any broker requesting a connection. The broker uses the certificate to establish an encrypted connection to the tunnel servlet. Once this connection is established, a secure connection between the client application and the tunnel servlet can be negotiated by the client application and the application server or Web server.

Enabling HTTP/HTTPS Support

The procedures for enabling HTTP and HTTPS support are essentially the same for both protocols, although a few extra steps are required in the HTTPS case to generate and access the needed encryption keys and certificates. The steps are as follows. (For HTTPS, start with step 1; for non-secure HTTP, start with step 4.)

1. *(HTTPS only)* Generate a self-signed certificate for the HTTPS tunnel servlet.

2. *(HTTPS only)* Modify the deployment descriptor in the tunnel servlet's `.war` file to specify the location and password of the certificate key store.

3. *(HTTPS only)* Validate the Web or application server's self-signed certificate and install it in the client application's trust store.

4. *(HTTP and HTTPS)* Deploy the HTTP or HTTPS tunnel servlet.

5. *(HTTP and HTTPS)* Configure the broker's `httpjms` or `httpsjms` connection service and start the broker.

6. *(HTTP and HTTPS)* Configure an HTTP or HTTPS connection.

The following subsections describe each of these steps in greater detail, using GlassFish Server as an example for purposes of illustration. If you are using a different application server or Web server (such as Oracle iPlanet Web Server), the procedures will be substantially similar but may differ in detail; see your server product's own documentation for specifics.

Step 1 (HTTPS Only): Generating a Self-Signed Certificate for the Tunnel Servlet

Message Queue's SSL support is oriented toward securing on-the-wire data, on the assumption that the client is communicating with a known and trusted server. Therefore, SSL is implemented using only self-signed server certificates. Before establishing an HTTPS connection, you must obtain such a certificate. (This step is not needed for ordinary, non-secure HTTP connections.)

Run the Message Queue Key Tool utility (`imqkeytool`) to generate a self-signed certificate for the tunnel servlet. (On UNIX systems, you may need to run the utility as the root user in order to have permission to create the key store.) Enter the following at the command prompt:

```
imqkeytool -servlet keyStoreLocation
```
where *keyStoreLocation* is the location of Message Queue's key store file.

The Key Tool utility prompts you for a key store password:

```
Enter keystore password:
```
After you have entered a valid password, the utility prompts you for identifying information from which to construct an X.500 distinguished name. Table C–1 shows the prompts and the values to be provided for each prompt. Values are case-insensitive and can include spaces.

Table C–1   Distinguished Name Information Required for a Self-Signed Certificate

Prompt	X.500 Attribute	Description	Example
`What is your first and last name?`	*commonName* (*CN*)	Fully qualified name of server running the broker	`mqserver.sun.com`
`What is the name of your organizational unit?`	*organizationalUnit* (*OU*)	Name of department or division	`purchasing`
`What is the name of your organization?`	*organizationName* (*ON*)	Name of larger organization, such as a company or government entity	`Acme Widgets, Inc.`
`What is the name of your city or locality?`	*localityName* (*L*)	Name of city or locality	`San Francisco`
`What is the name of your state or province?`	*stateName* (*ST*)	Full (unabbreviated) name of state or province	`California`
`What is the two-letter country code for this unit?`	*country* (*C*)	Standard two-letter country code	`US`

When you have entered the information, the Key Tool utility displays it for confirmation: for example,

```
Is CN=mqserver.sun.com, OU=purchasing, ON=Acme Widgets, Inc.,
L=San Francisco, ST=California, C=US correct?
```

To accept the current values and proceed, enter `yes`; to reenter values, accept the default or enter `no`. After you confirm, the utility pauses while it generates a key pair.

Next, the utility asks for a password to lock the key pair (key password). Press Return in response to this prompt to use the same password for both the key password and the key store password.

> **Caution:** Be sure to remember the password you specify. You must provide this password later to the tunnel servlet so it can open the key store.

The Key Tool utility generates a self-signed certificate and places it in Message Queue's key store file at the location you specified for the *keyStoreLocation* argument.

> **Caution:** The HTTPS tunnel servlet must be able to see the key store. Be sure to move or copy the generated key store from the location specified by *keyStoreLocation* to one accessible to the tunnel servlet (see Step 4 (HTTP and HTTPS): Deploying the Tunnel Servlet).

Step 2 (HTTPS Only): Specifying the Key Store Location and Password

The tunnel servlet's Web archive (`.war`) file includes a deployment descriptor, an XML file containing the basic configuration information needed by the application server or Web server to load and run the servlet. Before deploying the `.war` file for the HTTPS tunnel servlet, you must edit the deployment descriptor to specify the location and password of the certificate key store. (This step is not needed for ordinary, non-secure HTTP connections.)

To Specify the Location and Password of the Certificate Key Store

1. Copy the `.war` file to a temporary directory:

   ```
   cp IMQ_HOME/lib/imqhttps.war /tmp
   ```
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.

   Enter the command

   ```
   ls  -l WEB-INF/web.xml
   ```
   to confirm that the deployment descriptor file (`WEB-INF/web.xml`) was successfully extracted.

5. Edit the deployment descriptor to specify the key store location and password.

   Edit the `web.xml` file to provide appropriate values for the `keystoreLocation` and `keystorePassword` elements (as well as `servletPort` and `servletHost`, if necessary): for example,

   ```
   <init-param>
   <param-name>keystoreLocation</param-name>
   <param-value>/local/tmp/imqhttps/keystore</param-value>
   </init-param>
   <init-param>
   <param-name>keystorePassword</param-name>
   <param-value>shazam</param-value>
   </init-param>
   <init-param>
   <param-name>servletHost</param-name>
   <param-value>localhost</param-value>
   </init-param>
   ```

```
<init-param>
<param-name>servletPort</param-name>
<param-value>7674</param-value>
</init-param>
```

> **Note:** If you are concerned about exposure of the key store
> password, you can use file-system permissions to restrict access to
> the `imqhttps.war` file.)

6. Reassemble the contents of the `.war` file.

```
jar uvf  imqhttps.war  WEB-INF/web.xml
```

Step 3 (HTTPS Only): Validating and Installing the Server's Self-Signed Certificate

In order for a client application to communicate with the Web or application server,
you must validate the server's self-signed certificate and install it in the application's
trust store. The following procedure shows how:

To Validate and Install the Server's Self-Signed Certificate

1. Validate the server's certificate.

By default, the GlassFish Server generates a self-signed certificate and stores it in a
key store file at the location

appServerRoot`/glassfish/domains/domain1/config/keystore.jks`
where *appServerRoot* is the root directory in which the application server is
installed.

> **Note:** If necessary, you can use the JDK Key Tool utility to
> generate a key store of your own and use it in place of the default
> key store. For more information, see the details about establishing a
> secure connection using SSL in "Introduction to Security in the Java
> EE Platform" in *The Java EE 7 Tutorial*.

a. Make the directory containing the key store file your current directory.

For example, to use the Application Server's default key store file (as shown
above), navigate to its directory with the command

`cd` *appServerRoot*`/glassfish/domains/domain1/config`
where *appServerRoot* is, again, the root directory in which the application
server is installed.

b. List the contents of the key store file.

The Key Tool utility's `-list` option lists the contents of a specified key store
file. For example, the following command lists the Application Server's default
key store file (`keystore.jks`):

`keytool  -list  -keystore keystore.jks  -v`
The `-v` option tells the Key Tool utility to display certificate fingerprints in
human-readable form.

c. Enter the key store password.

The Key Tool utility prompts you for the key store file's password:

```
Enter keystore password:
```

By default, the key store password is set to `changeit`; you can use the Key Tool utility's `-storepasswd` option to change it to something more secure. After you have entered a valid password, the Key Tool utility will respond with output like the following:

```
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: slas
Creation date: Nov 13, 2007
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=helios, OU=Sun Java System Application Server, O=Sun
Microsystems,
L=Santa Clara, ST=California, C=US
Issuer: CN=helios, OU=Sun Java System Application Server, O=Sun
Microsystems,
L=Santa Clara, ST=California, C=US
Serial number: 45f74784
Valid from: Tue Nov 13 13:18:39 PST 2007 until: Fri Nov 10 13:18:39 PST
2017
Certificate fingerprints:
        MD5:  67:04:CC:39:83:37:2F:D4:11:1E:81:20:05:98:0E:D9
        SHA1: A5:DE:D8:03:96:69:C5:55:DD:E1:C4:13:C1:3D:1D:D0:4C:81:7E:CB
        Signature algorithm name: MD5withRSA
        Version: 1
```

   d. Verify the certificate's fingerprints.

   Obtain the correct fingerprints for the Application Server's self-signed certificate by independent means (such as by telephone) and compare them with the fingerprints displayed by the `keytool list` command. *Do not* accept the certificate and install it in your application's trust store unless the fingerprints match.

2. Export the Application Server's certificate to a certificate file.

   Use the Key Tool utility's `-export` option to export the certificate from the Application Server's key store to a separate certificate file, from which you can then import it into your application's trust store. For example, the following command exports the certificate shown above, whose alias is `slas`, from the Application Server's default key store (`keystore.jks`) to a certificate file named `slas.cer`:

```
keytool  -export  -keystore keystore.jks  -storepass changeit
-alias slas  -file slas.cer
```
   The Key Tool utility responds with the output

```
Certificate stored in file <slas.cer>
```

3. Verify the contents of the certificate file.

   If you wish, you can double-check the contents of the certificate file to make sure it contains the correct certificate:

   a. List the contents of the certificate file.

The Key Tool utility's `-printcert` option lists the contents of a specified certificate file. For example, the following command lists the certificate file `slas.cer` that was created in the preceding step:

```
keytool  -printcert  -file slas.cer  -v
```

Once again, the `-v` option tells the Key Tool utility to display the certificate's fingerprints in human-readable form. The resulting output looks like the following:

```
Owner: CN=helios, OU=Sun Java System Application Server, O=Sun
Microsystems,
L=Santa Clara, ST=California, C=US
Issuer: CN=helios, OU=Sun Java System Application Server, O=Sun
Microsystems,
L=Santa Clara, ST=California, C=US
Serial number: 45f74784
Valid from: Tue Nov 13 13:18:39 PST 2007 until: Fri Nov 10 13:18:39 PST
2017
Certificate fingerprints:
        MD5:  67:04:CC:39:83:37:2F:D4:11:1E:81:20:05:98:0E:D9
        SHA1: A5:DE:D8:03:96:69:C5:55:DD:E1:C4:13:C1:3D:1D:D0:4C:81:7E:CB
        Signature algorithm name: MD5withRSA
        Version: 1
```

    b. Confirm the certificate's contents.

       Examine the output from the `keytool printcert` command to make sure that the certificate is correct.

4. Import the certificate into your application's trust store.

The Key Tool utility's `-import` option installs a certificate from a certificate file in a specified trust store. For example, if your client application's trust store is kept in the file `/local/tmp/imqhttps/appKeyStore`, the following command will install the certificate from the file `slas.cer` created above:

```
keytool  -import  -file slas.cer  -keystore "/local/tmp/imqhttps/appKeyStore"
```

Step 4 (HTTP and HTTPS): Deploying the Tunnel Servlet

You can deploy the HTTP or HTTPS tunnel servlet on GlassFish Server either from the command line or by using the application server's Web-based administration GUI. In either case, you must then modify the Application Server's security policy file to grant permissions for the tunnel servlet.

To deploy the tunnel servlet from the command line, use the `deploy` subcommand of the application server administration utility (`asadmin`): for example,

```
asadmin deploy  --user admin  --passwordfile pfile.txt  --force=true
/local/tmp/imqhttps/imqhttps.war
```

The procedure below shows how to use the Web-based GUI to deploy the servlet.

After deploying the tunnel servlet (whether from the command line or with the Web-based GUI), proceed to Modifying the Application Server's Security Policy File for instructions on how to grant it the appropriate permissions.

To Deploy the HTTP or HTTPS Tunnel Servlet

1. Deploy the tunnel servlet:

    a. In the Web-based administration GUI, choose

```
App Server>Instances>appServerInstance>Applications>Web Applications
```

where *appServerInstance* is the application server instance on which you are deploying the tunnel servlet.

   b. Click the Deploy button.

2. Specify the `.war` file location:

   a. Enter the location of the tunnel servlet's Web archive file (`imqhttp.war` or `imqhttps.war`) in the File Path text field.

      The file is located in the `IMQ_HOME/lib` directory.

   b. Click the OK button.

3. Specify the context root directory:

   a. Enter the */contextRoot* portion of the tunnel servlet's URL.

      The URL has the form

```
http://hostName:portNumber/contextRoot/tunnel
```
or
```
https://hostName:portNumber/contextRoot/tunnel
```
      For example, if the URL for the tunnel servlet is

```
http://hostName:portNumber/imq/tunnel
```
      the value you enter would be

```
/imq
```
   b. Click the OK button.

      A confirmation screen appears, showing that the tunnel servlet has been successfully deployed and is enabled by default. The servlet is now available at the URL

```
http://hostName:portNumber/contextRoot/tunnel
```
or
```
https://hostName:portNumber/contextRoot/tunnel
```
      where *contextRoot* is the context root directory you specified in step a above. Clients can now use this URL to connect to the message service using an HTTP or HTTPS connection.

4. Modify the server's security policy file

    Once you have deployed the HTTP or HTTPS tunnel servlet, you must grant it the appropriate permissions by modifying the Application Server's security policy file, as described in the next procedure.

Modifying the Application Server's Security Policy File

Each application server instance has a security policy file specifying its security policies or rules. Unless modified, the default security policies would prevent the HTTP or HTTPS tunnel servlet from accepting connections from the Message Queue message broker. In order for the broker to connect to the tunnel servlet, you must add an additional entry to this policy file:

1. Open the security policy file.

    The file is named `server.policy` and resides at a location that varies depending on your operating system platform. On the Solaris platform, for example, the policy file for server `jeeves` would be located at

```
appServerRoot/glassfish/domains/domain1/jeeves/config/server.policy
```
    where *appServerRoot* is the root directory in which GlassFish Server is installed.

2. Add the following entry to the file:

```
grant codeBase
"file:appServerRoot/glassfish/domains/domain1/jeeves
/applications/j2ee-modules/imqhttps/-
{
permission java.net.SocketPermission "*","connect,accept,resolve";
};
```

3. Save and close the security policy file.

Step 5 (HTTP and HTTPS): Configuring the Connection Service

HTTP/HTTPS support is not activated for a broker by default, so before connecting using these protocols, you need to reconfigure the broker to activate the `httpjms` or `httpsjms` connection service. Table C–2 shows broker configuration properties pertaining specifically to these two connection services. Once reconfigured, the broker can be started normally, as described under Starting Brokers.

Table C–2   Broker Configuration Properties for the `httpjms` and `httpsjms` Connection Services

Property	Type	Default Value	Description
`imq.httpjms.http.servletHostimq.httpsjms.https.servletHost`	String	`localhost`	Host name or IP address of (local or remote) host running tunnel servlet

`imq.httpjms.http.servletPortimq.httpsjms.https.servletPort`	Integer	`httpjms:`
		`7675httpsjms:`
		`7674`
			Port number of tunnel servlet

`imq.httpjms.http.pullPeriodimq.httpsjms.https.pullPeriod`	Integer	-1	Interval, in seconds, between client HTTP/HTTPS requests If zero or negative, the client will keep one request pending at all times.
`imq.httpjms.http.connectionTimeoutimq.httpsjms.https.connectionTimeout`	Integer	60	Tunnel servlet timeout interval

To Activate the `httpjms` or `httpsjms` Connection Service

1. Open the broker's instance configuration file.

   The instance configuration file is named `config.properties` and is located in `IMQ_VARHOME/instances/instanceName/props`.

2. Add `httpjms` or `httpsjms` to the list of active connection services.

   Add the value `httpjms` or `httpsjms` to the `imq.service.activelist` property: for example,

   ```
   imq.service.activelist=jms,admin,httpjms
   ```
   or
   ```
   imq.service.activelist=jms,admin,httpsjms
   ```

3. Set any other HTTP/HTTPS-related configuration properties as needed.

   At startup, the broker looks for an application server or Web server and an HTTP or HTTPS tunnel servlet running on its local host machine. If necessary, you can reconfigure the broker to access a remote tunnel servlet instead, by setting the `servletHost` and `servletPort` properties appropriately (see Table C–2): for example,

   ```
   imq.httpjms.http.servletHost=helios
   imq.httpjms.http.servletPort=7675
   ```

You can also improve performance by reconfiguring the connection service's `pullPeriod` property. This specifies the interval, in seconds, at which each client issues HTTP/HTTPS requests to pull messages from the broker. With the default value of -1, the client will keep one such request pending at all times, ready to pull messages as fast as possible. With a large number of clients, this can cause a heavy drain on server resources, causing the server to become unresponsive. Setting the `pullPeriod` property to a positive value configures the client's HTTP/HTTPS transport driver to wait that many seconds between pull requests, conserving server resources at the expense of increased response times to clients.

The `connectionTimeout` property specifies the interval, in seconds, that the client runtime waits for a response from the HTTP/ HTTPS tunnel servlet before throwing an exception, as well as the time the broker waits after communicating with the tunnel servlet before freeing a connection. (A timeout is necessary in this case because the broker and the tunnel servlet have no way of knowing if a client that is accessing the tunnel servlet has terminated abnormally.)

Step 6 (HTTP and HTTPS): Configuring a Connection

To make HTTP/HTTPS connections to a broker, a client application needs an appropriately configured connection factory administered object. Before configuring the connection factory, clients wishing to use secure HTTPS connections must also have access to SSL libraries provided by the Java Secure Socket Extension (JSSE) and must obtain a trusted root certificate.

Installing a Root Certificate (HTTPS Only)

If the root certificate of the certification authority (CA) that signed your application server's (or Web server's) certificate is not in the trust store by default, or if you are using a proprietary application server or Web server certificate, you must install the root certificate in the trust store. (This step is not needed for ordinary, non-secure HTTP connections, or if the CA's root certificate is already in the trust store by default.)

Installing a Root Certificate in the Trust Store

1. Import the root certificate.

   Execute the command

   ```
   JRE_HOME/bin/keytool  -import   -trustcacerts
   -alias certAlias  -file certFile
   -keystore trustStoreFile
   ```
   where *certFile* is the file containing the root certificate, *certAlias* is the alias representing the certificate, and *trustStoreFile* is the file containing your trust store.

2. Confirm that you trust the certificate.

   Answer `YES` to the question `Trustthiscertificate?`

3. Identify the trust store to the client application.

   In the command that launches the client application, use the `-D` option to specify the following properties:

   ```
   javax.net.ssl.trustStore=trustStoreFile
   javax.net.ssl.trustStorePassword=trustStorePassword
   ```

Configuring the Connection Factory (HTTP and HTTPS)

To enable HTTP/HTTPS support, you need to set the connection factory's `imqAddressList` attribute to the URL of the HTTP/HTTPS tunnel servlet. The URL has the form

`http://`*hostName*`:`*portNumber*`/`*contextRoot*`/tunnel`
or

`https://`*hostName*`:`*portNumber*`/`*contextRoot*`/tunnel`
where *hostName*`:`*portNumber* is the host name and port number of the application server or Web server hosting the tunnel servlet and *contextRoot* is the context root directory you specified when deploying the tunnel servlet on the server, as described above under Step 4 (HTTP and HTTPS): Deploying the Tunnel Servlet.

You can set the `imqAddressList` attribute in any of the following ways:

- Use the `-o` option to the `imqobjmgr` command that creates the connection factory administered object (see Adding a Connection Factory).

- Set the attribute when creating the connection factory administered object using the Administration Console (`imqadmin`).

- Use the `-D` option to the command that launches the client application.

- Use an API call to set the attributes of the connection factory after you create it programmatically in client application code (see the *Open Message Queue Developer's Guide for Java Clients*).

Using a Single Servlet to Access Multiple Brokers (HTTP and HTTPS)

It is not necessary to configure multiple application or Web servers and tunnel servlets in order to access multiple brokers; you can share a single server instance and tunnel servlet among them. To do this, you must configure the `imqAddressList` connection factory attribute as follows:

`http://`*hostName*`:`*portNumber*`/`*contextRoot*`/tunnel?ServerName=`
brokerHostName`:`*instanceName*
or

`https://`*hostName*`:`*portNumber*`/`*contextRoot*`/tunnel?ServerName=`
brokerHostName`:`*instanceName*
where *brokerHostName* is the broker instance host name and *instanceName* is the name of the specific broker instance you want your client to access.

To check that you have entered the correct values for *brokerHostName* and *instanceName*, generate a status report for the HTTP/HTTPS tunnel servlet by accessing the servlet URL from a browser:

`http://localhost:8080/imqhttp/tunnel`
The report lists all brokers being accessed by the servlet, as shown in Example C–1.

Example C–1 Tunnel Servlet Status Report

```
HTTP tunnel servlet ready.
Servlet Start Time : Thu May 30 01:08:18 PDT 2002
Accepting secured connections from brokers on port : 7675
Total available brokers = 2
Broker List :
   helios:broker1
   selene:broker2
```

Using an HTTP Proxy

To use an HTTP proxy to access the HTTPS tunnel servlet, set the system properties `http.proxyHost` and `http.proxyPort` to the proxy server's host name and port number. You can set these properties using the `-D` option to the command that launches the client application.

Troubleshooting

This section describes possible problems with an HTTP or HTTPS connection and provides guidance on how to handle them.

Server or Broker Failure

The consequences of a server or broker failure in an (HTTP or HTTPS) connection vary depending on the specific component that has failed:

- If the application server or Web server fails and is restarted, all existing connections are restored with no effect on clients.

- If the broker fails and is restarted, an exception is thrown and clients must reestablish their connections.

- In the unlikely event that both the broker and the application server or Web server fail and the broker is not restarted, the application server or Web server will restore client connections and continue waiting for a broker connection without notifying clients. To avoid this situation, always restart the broker after a failure.

Client Failure to Connect Through the Tunnel Servlet

If an HTTPS client cannot connect to the broker through the tunnel servlet, do the following:

If a Client Cannot Connect

1. Start the tunnel servlet and the broker.

2. Use a browser to access the servlet manually through the tunnel servlet URL.

3. Use the following administrative commands to pause and resume the connection:

   ```
   imqcmd pause svc   -n httpsjms  -u admin
   imqcmd resume svc  -n httpsjms  -u admin
   ```
   When the service resumes, an HTTPS client should be able to connect to the broker through the tunnel servlet.

D

JMX Support

Message Queue includes support for Java-based client programs to programmatically configure and monitor Message Queue resources by means of the Java Management Extensions (JMX) application programming interface. These resources include brokers, connection services, connections, destinations, durable subscribers, and transactions, Use of the JMX API from the client side is fully described in the *Open Message Queue Developer's Guide for JMX Clients*. This appendix describes the JMX support infrastructure on the broker side, including the following topics:

- JMX Connection Infrastructure
- JMX Configuration

JMX Connection Infrastructure

The JMX API allows Java client applications to monitor and manage broker resources by programmatically accessing JMX MBeans (managed beans) that represent broker resources. As explained in "JMX-Based Administration" in *Open Message Queue Technical Overview*, the broker implements MBeans associated with individual broker resources, such as connection services, connections, destinations, and so forth, as well as with whole categories of resources, such as the set of all destinations on a broker. There are separate configuration MBeans and monitor MBeans for setting a resource's configuration properties and monitoring its runtime state.

MBean Access Mechanism

In the JMX implementation used by Message Queue, JMX client applications access MBeans using remote method invocation (RMI) protocols provided by JDK 1.5 (and later).
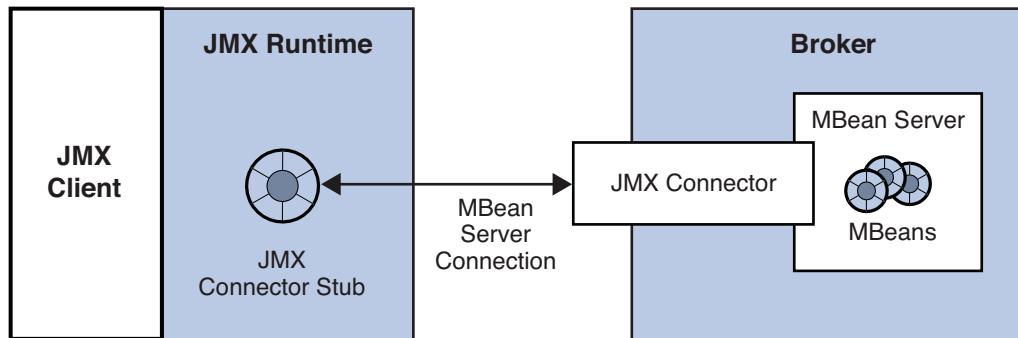
When a broker is started, it automatically creates MBeans that correspond to broker resources and places them in an MBean server (a container for MBeans). JMX client applications access the MBean server by means of an JMX RMI connector (heretofore called a JMX connector), which is used to obtain an MBean server connection, which, in turn, provides access to individual MBeans.

The broker also creates and configures two default JMX connectors, `jmxrmi` and `ssljmxrmi`. These connectors are similar to the broker connection services used to provide connections to the broker from JMS clients. By default, only the `jmxrmi` connector is activated at broker startup. The `ssljmxrmi` connector, which is configured to use SSL encryption, can be activated using the `imq.jmx.connector.activelist` broker property (see To Activate the SSL-Based JMX connector).

JMX client applications programmatically access JMX MBeans by first obtaining an MBean server connection from the `jmxrmi` or `ssljmxrmi` connector. The connector itself

is accessed by using a proxy object (or stub) that is obtained from the broker by the JMX client runtime, as shown in the following figure. Encapsulated in the connector stub is the port at which the connector resides, which is dynamically assigned each time a broker is started, and other connection properties.

Figure D–1   Basic JMX Infrastructure

The JMX Service URL

JMX client applications obtain a JMX connector stub using an address called the JMX service URL. The value and format of the JMX service URL depends on how the broker's JMX support is configured:

- **Static JMX service URL.** The JMX service URL specifies the location of the JMX connector stub in an RMI registry. When the broker is started, it creates the JMX connector stub and places it in the specified location in the RMI registry. This location is fixed across broker startups.

- **Dynamic JMX service URL.**The JMX service URL contains the JMX connector stub as a serialized object. This URL is dynamically created each time the broker is started.

A JMX service URL has the following form:
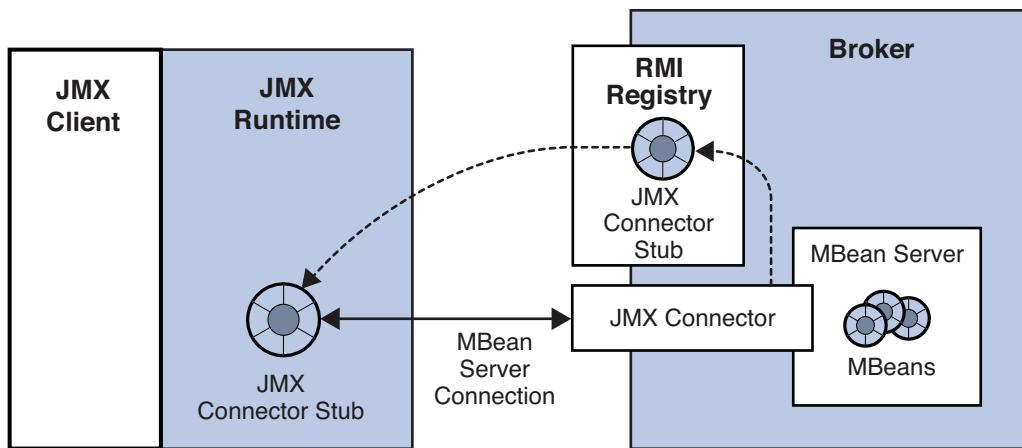
```
service:jmx:rmi://brokerHost[:connectorPort]urlpath
```

where `rmi://brokerHost[:connectorPort]` specifies the host (and optionally a port) used by the JMX connector. By default the port is assigned dynamically on broker startup, but can be set to a fixed value for JMX connections through a firewall.

The *urlpath* portion of the JMX service URL depends on whether the JMX service URL is static (see Static JMX Service URL: Using an RMI Registry) or dynamic (see Dynamic JMX Service URL: Not Using an RMI Registry). In either case, you can determine the value of the JMX service URL by using the `imqcmd list jmx` subcommand (see the examples in RMI Registry Configuration).

By default, the broker does not use an RMI registry, and the JMX runtime obtains a JMX connector stub by extracting it from a dynamic JMX service URL. However, if the broker is configured to use an RMI registry, then JMX runtime uses a static JMX service URL to perform a JNDI lookup of the JMX connector stub in the RMI registry. This approach, illustrated in the following figure, has the advantage of providing a fixed location at which the connector stub resides, one that does not change across broker startups.
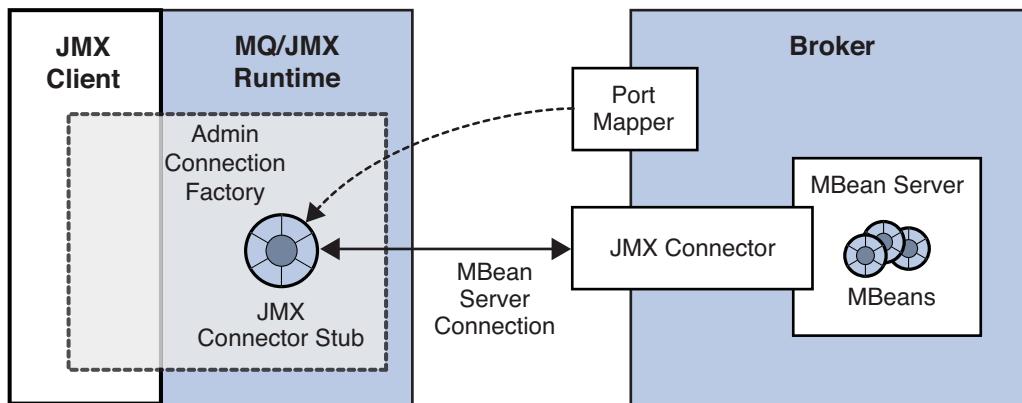
Figure D–2   Obtaining a Connector Stub from an RMI Registry

The Admin Connection Factory

Message Queue also provides, as a convenience, an `AdminConnectionFactory` class that hides the details of the JMX Service URL and JMX connector stub. The Admin Connection Factory uses the Message Queue Port Mapper service to get the relevant JMX Service URL (regardless of the form being used) and thereby obtain a JMX connector stub. JMX applications that use the Admin Connection Factory only need to know the broker's host and Port Mapper port. The scheme is shown in the following figure.

Figure D–3   Obtaining a Connector Stub from an Admin Connection Factory

For programmatic details, see "Obtaining a JMX Connector from an Admin Connection Factory" in *Open Message Queue Developer's Guide for JMX Clients*.

JMX Configuration

Broker configuration properties that support JMX are listed in Table 17–18. These properties can be set in the broker's instance configuration file (`config.properties`) or at broker startup with the `-D` option of the Broker utility (`imqbrokerd`). None of these properties can be set dynamically with the Command utility (`imqcmd`). In addition, as described below, some of these properties can be set with corresponding `imqbrokerd` options.

This section discusses several JMX configuration topics:

- [RMI Registry Configuration](#)
- [SSL-Based JMX Connections](#)
- [JMX Connections Through a Firewall](#)

RMI Registry Configuration

You can configure the broker to do any of the following:

- **Start an RMI registry** (`imq.jmx.rmiregistry.start=true`)

  If the broker is configured to start an RMI registry, then the broker will do the following:

  - Start an RMI registry in the broker process. The RMI registry will remain operational during the lifetime of the broker.
  - Store the JMX connector stub for it's connectors in this RMI registry.
  - Advertise a static JMX Service URL that points to the relevant JMX connector stub in this registry.
  - Shut down the RMI registry as part of the broker shutdown process.

- **Use an existing RMI registry** (`imq.jmx.rmiregistry.use=true`)

  If the broker is configured to use an existing RMI registry on the local host, then the broker will do the following:

  - Expect an RMI registry to be running on the same host (at a port which can also be specified)
  - Store the JMX connector stub for it's connectors in this externally managed RMI registry.
  - Advertise a static JMX Service URL that points to the relevant JMX connector stub in this registry. This means the registry must remain operational during the lifetime of the broker.
  - Not shut down the RMI registry as part of the broker shutdown process.

- **Not use a registry at all** (both `imq.jmx.rmiregistry.start` and `imq.jmx.rmiregistry.use` are set to `false`).

  If the broker is configured to not use a registry, then the broker will advertise a dynamic JMX Service URL that contains the JMX connector stub as a serialized object.

The choice of using or not using an RMI registry depends upon whether you want a static or dynamic JMX Service URL, respectively. The advantages and disadvantages of using an RMI registry are shown in the following table.

Table D–1   Advantages and Disadvantages of Using an RMI Registry

Scenario	Broker Configuration	Advantages	Disadvantages

Using a Registry (Static JMX Service URL)	Configuration Properties:
	`imq.jmx.rmigegistry.start`
	`imq.jmx.rmigegistry.use`
	`imq.jmx.rmigegistry.port`
		The value of the JMX Service URL is constant across broker restarts.	Broker depends on an RMI registry, either one it starts or one that is externally available. There is therefore one more port to worry about with regard to port conflicts or firewall configurations.

Table D–1   (Cont.) Advantages and Disadvantages of Using an RMI Registry

Scenario	Broker Configuration	Advantages	Disadvantages
Not Using a Registry (Dynamic JMX Service URL)	Default	Broker does not start up an RMI registry. There is therefore one less port to worry about with regard to port conflicts or firewall configurations.	The value of the JMX Service URL changes at every broker startup. JMX applications need to be provided a new URL every time the broker restarts. (This is not an issue with JMX client applications that use the `AdminConnectionFactory` class.)

If a registry is being used, the `imq.jmx.rmiregistry.port` property specifies the port number for the RMI registry. For convenience, you can also specify these RMI registry related properties by using equivalent Broker utility (`imqbrokerd`) options at broker startup: `-startRmiRegistry`, `-useRmiRegistry`, and `-rmiRegistryPort`, respectively (see Table 16–1).

Static JMX Service URL: Using an RMI Registry

When using an RMI Registry to store a JMX connector stub, the *urlpath* portion of the JMX service URL (see The JMX Service URL) does not change across broker startups and has the following form:

`/jndi/rmi://`*brokerHost*`[:`*rmiPort*`]/`*brokerHost*`/`*portMapperPort*`/`*connectorName*

This path consists of two segments:

- `/jndi/rmi://`*brokerHost*`[:`*rmiPort*`]`

   Specifies the RMI registry host and port at which the JMX contector stub is obtained by performing a JNDI lookup. The default port is 1099.

- `/`*brokerHost*`/`*portMapperPort*`/`*connectorName*

   Specifies the location within the RMI registry where the JMX connector stub is stored.

Example D–1   JMX Service URL When Using an RMI Registry

The following example shows the JMX service URL for the default `jmxrmi` connector in the case where an RMI registry is started on port 1098 on a host called `yourhost`:

```
# imqbrokerd -startRmiRegistry -rmiRegistryPort 1098

% imqcmd list jmx -u admin -passfile /myDir/psswds
Listing JMX Connectors on the broker specified by:

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


Name      Active URL
jmxrmi    true    service:jmx:rmi://yourhost/jndi/rmi://yourhost:1098
                  /yourhost/7676/jmxrmi
ssljmxrmi false

Successfully listed JMX Connectors.
```

The JMX service URL could potentially contain a hostname and port three separate times, indicating the location of the JMX connector, the RMI registry, and the broker, respectively.

Dynamic JMX Service URL: Not Using an RMI Registry

When not using an RMI Registry to store a JMX connector stub, the *urlpath* portion of the JMX service URL is dynamically generated at broker startup and has the following form:

```
/stub/rO0ABdmVyLlJlpIDJyGvQkwAAAARod97VdgAEAeA==
```

where the string following `/stub/` is the is the serialized JMX connector stub encoded in BASE64 (shortened above for legibility)

Example D–2   JMX Service URL When Not Using an RMI Registry

The following example shows the JMX service URL for the default `jmxrmi` connector when no RMI registry is started by the broker and no existing registry is used.

```
# imqbrokerd

% imqcmd list jmx -u admin -passfile /myDir/psswds
Listing JMX Connectors on the broker specified by:

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


Name      Active  URL
jmxrmi    true    service:jmx:rmi://yourhost/stub/rO0ABdmVyLlJlpIDJy==

ssljmxrmi false

Successfully listed JMX Connectors.
```

SSL-Based JMX Connections

If you need to have secure, encrypted connections between a JMX client and the broker's MBean server, then you need to configure both sides of the connection accordingly.

Broker Side SSL Configuration

As mentioned in JMX Connection Infrastructure, a broker is configured by default for non-secure communication using the preconfigured `jmxrmi` connector. Applications wishing to use the Secure Socket Layer (SSL) for secure communication must activate the alternate `ssljmxrmi` connector. The `ssljmxrmi` connector is preconfigured with `imq.jmx.connector.`*RMIconnectorName*`.useSSL=true`.

To Activate the SSL-Based JMX connector

1. Obtain and install a signed certificate.

    The procedure is the same as for the `ssljms`, `ssladmin`, or `cluster` connection service, as described under Using Signed Certificates.

2. Install the root certification authority certificate in the trust store if necessary.

3. Add the `ssljmxrmi` connector to the list of JMX connectors to be activated at broker startup:

   ```
   imq.jmx.connector.activelist=jmxrmi,ssljmxrmi
   ```

4. Start the broker.

   Use the Broker utility (`imqbrokerd`), either passing it the keystore password in a passfile or typing it from at the command line when prompted.

5. Disable validation of certificates if desired.

   By default, the `ssljmxrmi` connector (or any other SSL-based connector) is configured to validate all broker SSL certificates presented to it. Validation will fail if the signer of the certificate is not in the client's trust store. To avoid this validation (for instance, when using self-signed certificates during software testing), set the broker property `imq.jmx.connector.ssljmxrmi.brokerHostTrusted` to `true`.

JMX Client Side SSL Configuration

On the client side, if the `AdminConnectionFactory` class is being used to obtain a JMX connector, the `AdminConnectionFactory` object must be configured with a URL specifying the `ssljmxrmi` connector:

```
AdminConnectionFactory  acf = new AdminConnectionFactory();
   acf.setProperty(AdminConnectionConfiguration.imqAddress,
                   "mq://myhost:7676/ssljmxrmi");
```

In addition, if the JMX client needs to access the trust store, use the system properties `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` to point the JMX client to the trust store. For example:

```
java -Djavax.net.ssl.trustStore=/tmp/myStrustsore
-Djavax.net.ssl.trustStorePassword=myTurstword MyApp
```

JMX Connections Through a Firewall

If a JMX client application needs to connect to a broker that is located behind a firewall, the broker must be configured to use fixed JMX ports so the firewall can, in turn, be configured to allow traffic on these ports. The relevant ports are the following:

- The port used by the JMX connector. The property used to configure this port is `imq.jmx.connector.`*connectorName*`.port`, where *connectorName* can be `jmxrmi` or `ssljmxrmi`.

- The port used by the RMI registry, if any. The property used to configure this port is `imq.jmx.rmiregistry.port`. The equivalent command line option for `imqbrokerd` is `-rmiRegistryPort`.

Once these ports are specified, configure the firewall to allow traffic on these ports.

Example D–3   JMX Configuration for Firewall When Not Using a RMI Registry

The following example starts a broker with no RMI registry and a `jmxrmi` connector on port 5656 on a host called `yourhost`, as follows:

```
# imqbrokerd -Dimq.jmx.connector.jmxrmi.port=5656
```

The resulting JMX service URL is:

```
service:jmx:rmi://yourhost:5656/stub/rO0ABdmVyLlJlpIDJy==
```

The JMX service URL shows the connector port. In this case, you need to configure the firewall to allow traffic only on port 5656.

Example D–4   JMX Configuration for Firewall When Using an RMI Registry

The following example starts a broker with an RMI registry on port 1098 and a `jmxrmi` connector on port 5656 on a host called `yourhost`, as follows:

```
# imqbrokerd -startRmiRegistry -rmiRegistryPort 1098
-Dimq.jmx.connector.jmxrmi.port=5656
```

The resulting JMX service URL is:

```
 service:jmx:rmi://yourhost:5656/jndi/rmi://yourhost:1098
                  /yourhost/7676/jmxrmi
```

The JMX service URL shows both these ports. You need to configure the firewall to allow traffic on ports 1098 and 5656.

E

Frequently Used Command Utility Commands

This appendix lists some frequently used Message Queue Command utility (`imqcmd`) commands. For a comprehensive list of command options and attributes available to you from the command line, refer to Command Utility in Command Utility.

Syntax

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

`-H` or `-h` provides comprehensive help. The `-v` subcommand provides version information.

When you use `imqcmd`, the Command utility prompts you for a password. To avoid the prompt (and to increase security), you can use the `-passfile` *pathToPassfile* option to point the utility to a password file that contains the administrator user name and password.

Example: `imqcmd query bkr -u` *adminUserName* `-passfile` *pathToPassfile* `-b` *myServer*`:7676`

Broker and Cluster Management

```
imqcmd query bkr
imqcmd pause bkr
imqcmd restart bkr
imqcmd resume bkr
imqcmd shutdown bkr -b myBroker:7676
imqcmd update bkr -o "imq.system.max_count=1000"
imqcmd reload cls
```

Broker Configuration Properties (-o option)

Broker Configuration Properties (`-o` option) lists frequently used broker configuration properties. For a full list of broker configuration properties and their descriptions, see Broker Properties Reference

Table E–1   Broker Configuration Properties (-o option)

Property	Notes
imq.autocreate.queue	
imq.autocreate.queue.maxNumActiveConsumers	Specify -1 for unlimited
imq.autocreate.queue.maxNumBackupConsumers	Specify -1 for unlimited
imq.autocreate.topic	
imq.cluster.url	
imq.destination.DMQ.truncateBody	
imq.destination.logDeadMessages	
imq.log.file.rolloverbytes	Specify -1 for unlimited
imq.log.file.rolloversecs	Specify -1 for unlimited
imq.log.level	NONEERRORWARNINGINFO
imq.message.max_size	Specify -1 for unlimited
imq.portmapper.port	
imq.system.max_count	Specify -1 for unlimited
imq.system.max_size	Specify -1 for unlimited

Service and Connection Management

```
imqcmd list svc
imqcmd query svc
imqcmd update svc -n jms -o "minThreads=200" -o "maxThreads=400" -o "port=8995"
imqcmd pause svc -n jms
imqcmd resume svc -n jms
imqcmd list cxn -svn jms
imqcmd query cxn -n 1234567890
```

Durable Subscriber Management

```
imqcmd list dur -d MyTopic
imqcmd destroy dur -n myDurSub -c "clientID-111.222.333.444"
imqcmd purge dur -n myDurSub -c "clientID-111.222.333.444"
```

Transaction Management

```
imqcmd list txn
imqcmd commit txn -n 1234567890
imqcmd query txn -n 1234567890
imqcmd rollback txn -n 1234567890
```

Destination Management

```
imqcmd create dst -n MyQueue -t q -o "maxNumMsgs=1000" -o "maxNumProducers=5"
imqcmd update dst -n MyTopic -t t -o "limitBehavior=FLOW_CONTROL| REMOVE_
OLDEST|REJECT_NEWEST|REMOVE_LOW_PRIORITY"
imqcmd compact dst -n MyQueue -t q
```

```
imqcmd purge dst -n MyQueue -t q
imqcmd pause dst -n MyQueue -t q -pst PRODUCERS|CONSUMERS|ALL
imqcmd resume dst -n MyQueue -t q
imqcmd destroy dst -n MyQueue -t q
imqcmd query dst -n MyQueue -t q
imqcmd list dst -tmp
```

Destination Configuration Properties (-o option)

Destination Configuration Properties (-o option) lists frequently used destination configuration properties. For a full list of destination configuration properties and their descriptions, see Physical Destination Property Reference

Table E–2   Destination Configuration Properties (-o option)

Property	Notes
consumerFlowLimit	Specify 0 or -1 for unlimited
isLocalOnly (create only)	
limitBehavior	FLOW_CONTROLREMOVE_OLDESTREJECT_NEWESTREMOVE_LOW_PRIORITY
localDeliveryPreferred (queue only)	
maxNumActiveConsumers (queue only)	Specify -1 for unlimited
maxNumBackupConsumers (queue only)	Specify -1 for unlimited
maxBytesPerMsg	Specify -1 for unlimited
maxNumMsgs	Specify -1 for unlimited
maxNumProducers	Specify -1 for unlimited
maxTotalMsgBytes	Specify -1 for unlimited
useDMQ	

Metrics

```
imqcmd metrics bkr -m cxn|rts|ttl -int 5 -msp 20
imqcmd metrics svc -m cxn|rts|ttl
imqcmd metrics dst -m con|dsk|rts|ttl
```