Administrator's Guide

iPlanet™ Message Queue for Java™

Version 2.0

August 24, 2001

Contents

List of Figures

List of Tables

List of Procedures

Preface

This book is a new, iPlanet Message Queue for Java (iMQ) 2.0 *Administrator's Guide*. It provides the background and information needed to perform administration tasks for an iMQ messaging system.

This preface contains the following sections:

Audience for This Guide

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

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

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

Organization of This Guide

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

Table 1 Book Contents

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

Conventions

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

Text Conventions

Table 2 Document Conventions

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

Environment Variable Conventions

iMQ makes use of two environment variables.

Table 3 iMQ Environment Variables

Environment Variable	Description
JMQ_HOME	This is the root iMQ installation directory in which all installed files are placed. On Windows, the installer sets JMQ_HOME to the iMQ installation directory. On Solaris and Linux, JMQ_HOME is manually set to /opt/SUNWjmq.
JMQ_VARHOME	This is a directory in which all transient or dynamically-created configuration and data files are stored. On Windows JMQ_VARHOME is set to JMQ_HOME\var. On Solaris and Linux, JMQ_VARHOME is manually set to /var/opt/SUNWjmq.

In this guide, JMQ_HOME and JMQ_VARHOME are shown *without* platform-specific environment variable notation or syntax (for example, $JMQ_HOME on UNIX). However, all path names use UNIX file separator notation (/).

Other Documentation Resources

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

The iMQ Documentation Set

The following documents are included with the iMQ product, listed in Table 4 in the order in which you would normally use them:

Table 4 iMQ Documentation Set

Document	Audience	Description
iMQ Installation Guide	Developers and administrators	Explains how to install iMQ software on Solaris, Linux, and Windows NT platforms.
iMQ Release Notes	Developers and administrators	Includes descriptions of new features, limitations, and known bugs, as well as technical notes.
iMQ Migration Guide	Developers and administrators	Explains differences between JMQ 1.1 and iMQ 2.0 and how to perform necessary conversions.
iMQ Developer's Guide	Developers	Provides a quick-start tutorial and programming information relevant to the iMQ implementation of JMS.
iMQ Administrator's Guide	Administrators, also recommended for developers	Provides background and information needed to perform administrative tasks using iMQ administration tools.

Online Help

iMQ 2.0 includes command line utilities for administering the iMQ message service. To access the online help for these utilities, see "Common Utility Options" on page 75.

iMQ 2.0 also includes a graphical user interface (GUI) administration tool, the iMQ Administration Console (`jmqadmin`). Context sensitive online help is included in the Administration Console.

JavaDoc

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

```
JMQ_HOME/doc/en/apidoc/index.html
```

This documentation can be viewed in any HTML browser such as Netscape or Internet Explorer. It includes standard JMS API documentation as well as iMQ-specific APIs for iMQ administered objects (see "iMQ Administered Objects" on page 66). The JavaDoc is mainly useful to developers of messaging applications.

Example Client Applications

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

```
JMQ_HOME/examples/jms
```

See the README file located in that directory.

The Java Message Service (JMS) Specification

The JMS 1.0.2 specification that iMQ implements can be found at the following location:

```
JMQ_HOME/doc/en/jmsspec/jms1_0_2-spec.pdf
```

Chapter 1

Overview

This chapter provides an overall introduction to iMQ and is of interest to both administrators and programmers.

What Is iMQ?

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

The JMS specification describes a set of programming interfaces (see "The JMS Programming Model" on page 27)—which provide a common way for Java applications to create, send, receive, and read messages in a distributed environment.

With iPlanet Message Queue for Java software, processes running on different platforms and operating systems can connect to a common iMQ Message Service (see "Message Service Architecture" on page 25) to send and receive information. Application developers are free to focus on the business logic of their applications, rather than on the low-level details of how their applications communicate across a network.

iMQ has features which go beyond the minimum requirements of the JMS specification. Among these features are the following:

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

Scalable Message Service Provides you the ability to add increasing numbers of client applications by balancing the load among a number of iMQ Message Service components working in tandem.

Tunable Performance Lets you increase performance of the iMQ Message Service when less reliability of delivery is acceptable.

Multiple Transports Supports the ability to communicate with an iMQ Message Service over a number of different transports, including TCP and HTTP, and using secure (SSL) connections.

JNDI support Supports both file-based and LDAP directory services as object stores and user repositories.

See the *iMQ 2.0 Release Notes* for documentation of JMS compliance-related issues.

iMQ Product Editions

The iPlanet Message Queue for Java product is available in three editions—trial, developer, and enterprise—each corresponding to a different licensed capacity, as described below. (To Upgrade iMQ from one edition to another, see the instructions in the *iMQ Installation Guide*.)

Trial Edition This edition is for product evaluation purposes. It has a 90-day maximum duration license (expiration is enforced by the software). The license places no limit on the number of broker components that implement the iMQ Message Service nor on the number of client connections supported by each broker. You cannot use the Trial edition for deploying and running messaging applications in a production environment.

Developer Edition This edition is for application development purposes. It has an unlimited duration license. The license limits the number of brokers implementing an iMQ Message Service to three, and the number of client connections supported by each broker to five. The Developer edition also includes the Trial edition license; once you have developed and debugged a messaging application, you can use the Trial edition license to perform unlimited-connection load testing for a maximum 90-day period. (For information on how to switch to the Trial edition license, see the `license` command line option described in "Starting a Broker" on page 109). You cannot use the Developer edition for deploying and running messaging applications in a production environment.

Enterprise Edition This edition is for deploying and running messaging applications in a production environment. You can also use the Enterprise edition for developing, debugging, and load testing messaging applications. It has an unlimited duration license. The license places no limit on the number of brokers implementing an iMQ Message Service nor on the number of client connections supported by each broker. However the license limits the Message Service to one host and one CPU; An additional license is needed for each broker running on an additional host or CPU.

Enterprise Messaging Systems

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

Requirements of Enterprise Messaging Systems

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

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

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

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

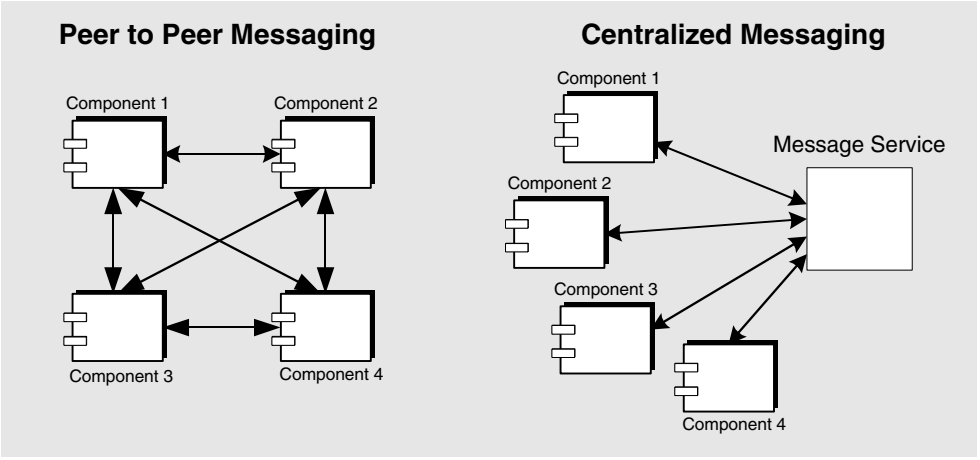
Scalability The messaging system must be able to accommodate increasing loads—increasing numbers of users and increasing numbers of messages—without a substantial loss of performance or message throughput. As businesses and applications expand, this becomes a very important requirement.

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

Centralized vs. Peer to Peer Messaging

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

Figure 1-1 Centralized vs. Peer to Peer Messaging



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

The preferred approach for enterprise messaging is a centralized messaging system, also illustrated in Figure 1-1. In this approach each messaging component maintains a connection to one central message service. The message service provides for routing and delivery of messages between components, and is

responsible for reliable delivery and security. Components interact with the message service through a well-defined programming interface. As components are added to the system, the number of connections rises only linearly, making it easier to scale the system by scaling the message service. In addition, the central message service provides for centralized management of the system.

Messaging System Concepts

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

Message

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

Message Service Architecture

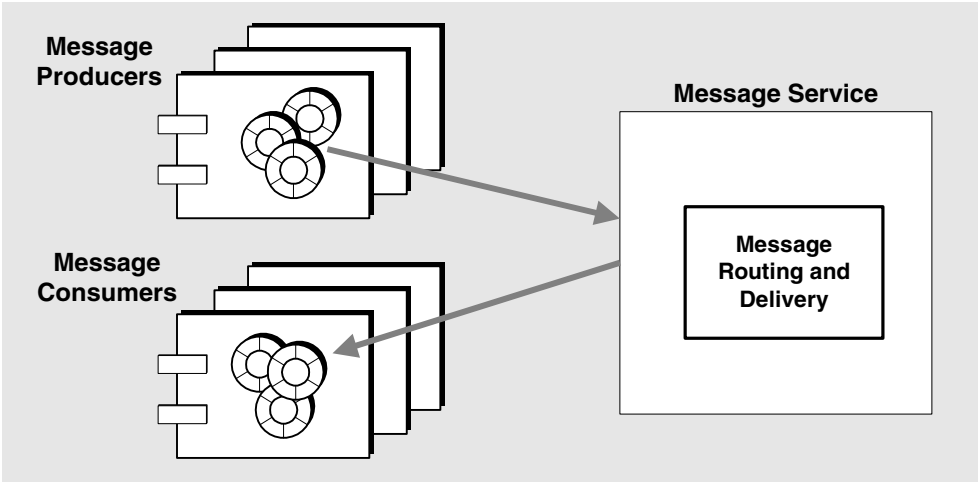
The basic architecture of a messaging system is illustrated in Figure 1-2. It consists of message producers and message consumers that exchange messages by way of a common message service. Any number of message producers and consumers can reside in the same messaging component (often called a client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

Messaging Styles

Messaging styles describe the relationship between producers and consumers, and include one to one, one to many, and many to many relationships. For example, you might have messages delivered from:

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

Figure 1-2 Message Service Architecture



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

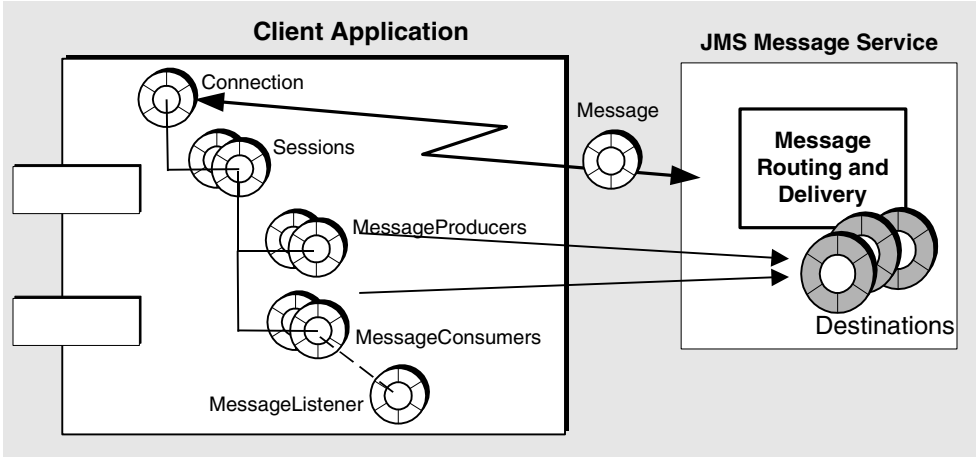
Historically, messaging systems supported various combinations of the above messaging styles. The Java Message Service (JMS) API was intended to create a common programming model of Java messaging applications. It supports all four messaging styles, implemented through both the point to point and publish/subscribe domains (see "Programming Domains" on page 30).

The JMS Programming Model

This section describes the programming model of the JMS specification. Because iMQ provides an implementation of JMS interfaces, JMS concepts are fundamental to understanding how an iMQ messaging system works. This introduction is meant to provide concepts and terminology needed to understand the remaining chapters of this book.

In the JMS programming model, client applications interact using a JMS application programming interface (API) to send and receive messages. This section introduces the objects that implement the JMS API and that are used to set up a client application for delivery of messages (for more information, see the *iMQ Developer's Guide*). The main interface objects are shown in Figure 1-3 and described in the following paragraphs.

Figure 1-3 JMS Programming Objects



Message

In the iMQ product, data is exchanged using JMS messages—messages that conform to the JMS specification. According to the JMS specification, a message is composed of three parts: a header, properties (which can be thought of as an extension of the header), and a body.

Properties are optional—they provide values that client applications can use to filter messages. A body is also optional—it contains the actual data to be exchanged.

Destination

A `Destination` is a JMS administered object (see "iMQ Administered Objects" on page 66) that identifies a *physical* destination in a JMS message service. A physical destination is a JMS message service entity to which producers send messages and from which consumers receive messages. The message service provides the routing and delivery for messages sent to a physical destination. A `Destination` administered object encapsulates provider-specific naming conventions for physical destinations. This lets client applications be provider independent.

ConnectionFactory

A `ConnectionFactory` is a JMS administered object (see "iMQ Administered Objects" on page 66) that encapsulates provider-specific connection configuration information. A client application uses it to create a connection over which messages are delivered. JMS administered objects can either be acquired through a Java Naming and Directory Service (JNDI) lookup or directly instantiated using provider-specific classes.

Connection

A `Connection` is a client application's active connection to a JMS message service. Both allocation of communication resources and authentication of a client take place when a connection is created. Hence it is a relatively heavy-weight object, and most client applications do all their messaging with a single connection. A connection is used to create sessions.

Session

A `Session` is a single-threaded context for producing and consuming messages. While there is no restriction on the number of threads that can use a session, the session should not be used *concurrently* by multiple threads. It is used to create the message producers and consumers that send and receive messages, and defines a serial order for the messages it delivers. A session supports reliable delivery through a number of acknowledgement options or by using transactions. A transacted session can combine a series of sequential operations into a single transaction that can span a number of producers and consumers.

Message Producer

A client application uses a `MessageProducer` to send messages to a physical destination. A `MessageProducer` object is normally created by passing a `Destination` administered object to a session's methods for creating a message producer. (If you create a message producer that does not reference a specific destination, then you must specify a destination for each message you produce.) A client can specify a default delivery mode, priority, and time-to-live for a message producer that govern all messages sent by a producer, except when explicitly over-ridden.

Message Consumer

A client application uses a `MessageConsumer` to receive messages from a physical destination. It is created by passing a `Destination` administered object to a session's methods for creating a message consumer. A message consumer can have a message selector that allows the message service to deliver only those messages to the message consumer that match the selection criteria. A message consumer can support either synchronous or asynchronous consumption of messages (see the *iMQ Developer's Guide*).

Message Listener

A client application uses a `MessageListener` object to consume messages asynchronously. The MessageListener is registered with a message consumer. A client application consumes a message when a session thread invokes the `onMessage()` method of the `MessageListener` object.

JMS Messaging Issues

This section describes a number of issues involved in using the JMS programming model that impact the administration of an iMQ messaging system. The discussion focuses on concepts and terminology that are needed to administer an iMQ Message Service.

Programming Domains

While JMS client programming is based on a common set of messaging concepts, two distinct message delivery models are supported by JMS—point to point and publish/subscribe.

Point to Point—Queue Destinations A message is delivered from a producer to one consumer. In this delivery model, the destination is a *queue*. Messages are first delivered to the queue destination, then delivered from the queue, one at a time, depending on the queue's delivery policy (see "Queue Destinations" on page 56), to one or more consumers registered for the queue. Any number of producers can send messages to a queue destination, but each message is guaranteed to be delivered to—and successfully consumed by—only *one* consumer. If there are no consumers registered for a queue destination, the queue holds messages it receives, and delivers them when a consumer registers for the queue.

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

These two message delivery models are handled using different sets of API objects—with somewhat different semantics—representing two programming domains, as shown in Table 1-1. To program with point to point messaging, you use the point to point domain objects, and to program with publish/subscribe messaging, you use the publish/subscribe domain objects.

Table 1-1 API Objects for JMS Programming Domains

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

JMS Provider Independence

In order to support the development of client applications that are portable to other JMS providers, iMQ supports what are called administered objects (see "iMQ Administered Objects" on page 66). Administered objects allow a client application to use logical names to look up and reference provider-specific objects. In this way client code does not need to know specific naming or addressing syntax or configurable properties used by a provider. This makes the code provider-independent.

Administered objects are iMQ system objects created and configured by an iMQ administrator. These objects are placed in a JNDI directory service, and a client application acquires them using a JNDI lookup.

iMQ administered objects can also be instantiated by the client, rather than looked up in a JNDI directory service. This has the drawback of requiring the application developer to use JMS provider-specific APIs. It also undermines the ability of an iMQ administrator to successfully control and manage an iMQ Message Service.

iMQ administered objects include `ConnectionFactory` objects used to create connections and `Destination` objects used to identify physical destinations when producing and consuming messages.

Client Identifiers

JMS providers must support the notion of a *client identifier*, which associates a client application's connection to a message service with a state maintained by the message service on behalf of the client. By definition, a client identifier can only be in use by one user at a time. Client identifiers are used in combination with a durable subscription name (see "Publish/Subscribe—Topic destinations" on page 30) to make sure that each durable subscription corresponds to only one user.

The JMS specification allows client identifiers to be set by the client application through an API method call, but recommends setting it administratively using a connection factory administered object (see "ConnectionFactory" on page 28). If hard wired into a connection factory, however, each user would need an individual connection factory to maintain a unique identity.

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

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

By default, if no other means of setting the client identifier has been chosen, the iMQ client runtime (see "iMQ Client Runtime" on page 63) chooses to use the IP address of the client host to set the client identifier, if needed. This default mechanism is there solely for the ease of demonstrating durable subscriber examples and is not recommended for deployed applications.

For deployed applications, the client identifier must either be programmatically set by the client application, using the JMS API, or administratively configured in the `ConnectionFactory` objects used by the client application.

Reliable Messaging

JMS defines two *delivery modes*:

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

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

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

Transactions/Acknowledgements

Reliable messaging depends on guaranteeing that delivery of persistent messages to and from a destination succeed. This reliability can be achieved using either of two mechanisms supported by an iMQ session: transactions or acknowledgements.

If a session is configured as *transacted*, then the production and/or consumption of one or more messages can be grouped into an atomic unit—a *transaction*. The client can commit the transaction if delivery of all messages succeeds. However, if an exception occurs on the commit operation, all operations in the transaction will be rolled back.

The scope of an iMQ transaction is always a single session: an iMQ transaction cannot span more than one client session. (In other words, the delivery of a message to a destination and the subsequent delivery of the message to a client cannot be placed in a single transaction.)

A session can also be configured as *non-transacted*, in which case it does not support transactions. Instead the session uses acknowledgements to assure reliable delivery.

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

Persistent Storage

The other important aspect of reliability is assuring that once persistent messages are delivered to their destinations, the message service does not lose them before they are delivered to consumers. This means that upon delivery of a persistent message to its destination, the message service must place it in a persistent data store (see "Persistence Manager" on page 46). If the message service goes down for any reason, it can recover the message and deliver it to the appropriate consumers. While this adds overhead to message delivery, it also adds reliability.

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

Messaging applications that are concerned about guaranteeing delivery of persistent messages must either employ queue destinations or employ durable subscriptions to topic destinations.

Performance Trade-offs

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

Message Selection

JMS provides a mechanism by which a message service can perform message filtering and routing based on criteria placed in message selectors. A producing client can place application-specific properties in the message, and a consuming client can indicate its interest in messages using selection criteria based on such properties. This simplifies the work of the client and eliminates the overhead of delivering messages to clients that don't need them. However it adds some additional overhead to the message service processing the selection criteria. Message selector syntax and semantics are outlined in the JMS specification (see "The Java Message Service (JMS) Specification" on page 20).

Message Order and Priority

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

Beyond this, the ordering of messages consumed by a client application can have only a rough relationship to the order in which they were produced. This is because the delivery of messages to destinations and the delivery from those destinations can depend on a number of issues that affect timing, such as the order in which the messages are sent, the sessions (connections) from which they are sent, whether the messages are persistent, the lifetime of the messages, the priority of the messages, the message delivery policy of queue destinations (see "Queue Destinations" on page 56), and message service availability.

In the case of an iMQ Message Service using multiple interconnected brokers (see "Multi-Broker Configurations (Clusters)" on page 58) the ordering of messages consumed by a client is further complicated by the fact that the order of delivery from destinations on different brokers is indeterminant. Hence, a message delivered by one broker might precede a message delivered by another broker even though the latter might have received the message first.

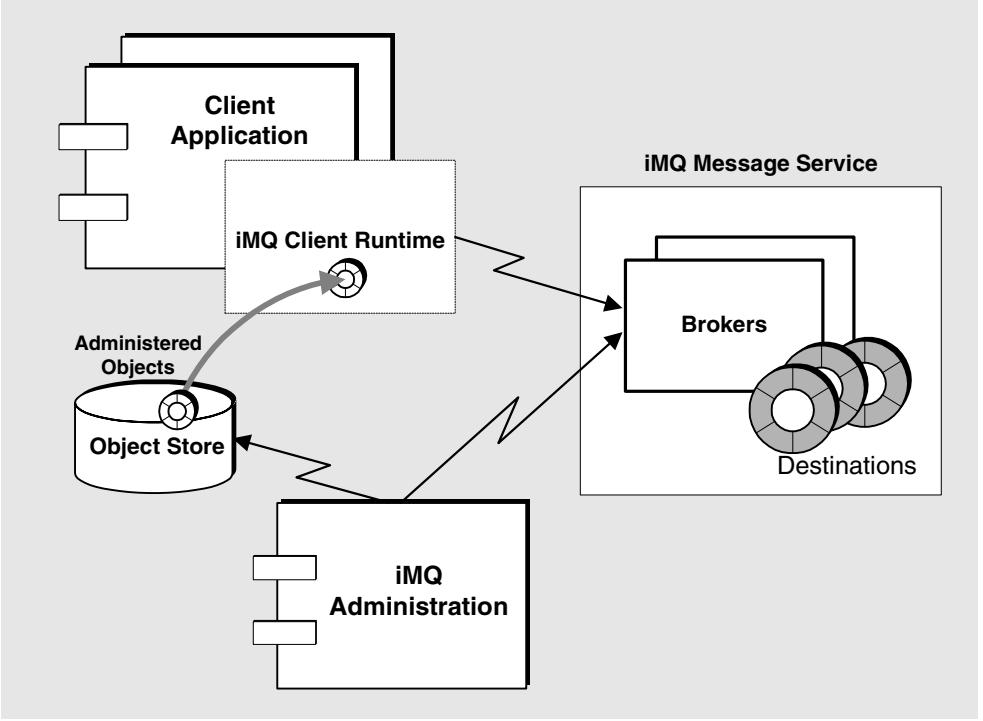
In any case, for a given consumer, precedence is given for higher priority messages over lower priority messages.

Chapter 2

The iMQ Messaging System

This chapter describes the iMQ messaging system, with specific attention to the main parts of the system, as illustrated in Figure 2-1, and how they work together to provide for reliable message delivery.

Figure 2-1 iMQ System Architecture



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

- iMQ Message Service
- iMQ Client Runtime
- iMQ Administered Objects
- iMQ Administration

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

iMQ Message Service

This section describes the different parts of the iMQ Message Service shown in Figure 2-1 on page 37. These include:

Brokers A broker provides delivery services for an iMQ messaging system. Message delivery relies upon a number of supporting components that handle connection services, message routing and delivery, persistence, security, and logging (see "Brokers" on page 38 for more information). A Message Service can employ one or more brokers (see "Multi-Broker Configurations (Clusters)" on page 58).

Physical Destinations Delivery of a message is a two-phase process—delivery from a producing client to a physical destination maintained by a broker, followed by delivery from the destination to one or more consuming clients. Physical destinations represent locations in a broker's physical memory and/or persistent storage (see "Physical Destinations" on page 55 for more information).

Brokers

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

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

Figure 2-2 Broker Components

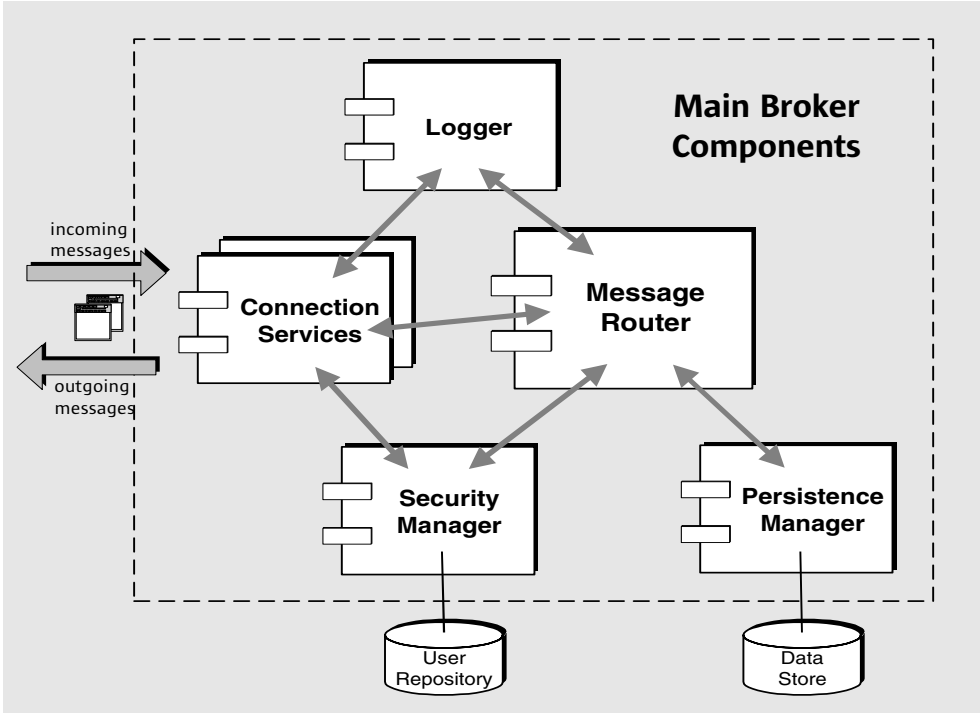


Table 2-1 Main Broker Components and Function

Component	Description/Function	See Page
Connection Service	Manages the physical connections between a broker and clients, providing transport for incoming and outgoing messages.	page 40
Message Router	Manages the routing and delivery of messages: These include JMS messages as well as control messages used by the iMQ messaging system to support JMS message delivery.	page 42

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

Component	Description/Function	See Page
Persistence Manager	Manages the writing of data to persistent storage so that system failure does not result in failure to deliver JMS messages.	page 46
Security Manager	Provides authentication services for users requesting connections to a broker and authorization services (access control) for authenticated users.	page 48
Logger	Writes monitoring and diagnostic information to log files or the console so that an administrator can monitor and manage a broker.	page 53

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

Connection Services

An iMQ broker supports communication with both application (JMS) clients and iMQ administration clients. Each service is specified by its service type and connection type.

service type whether it provides JMS message delivery (`NORMAL`) or broker administration (`ADMIN`) services

connection type the underlying transport protocol layer that supports it.

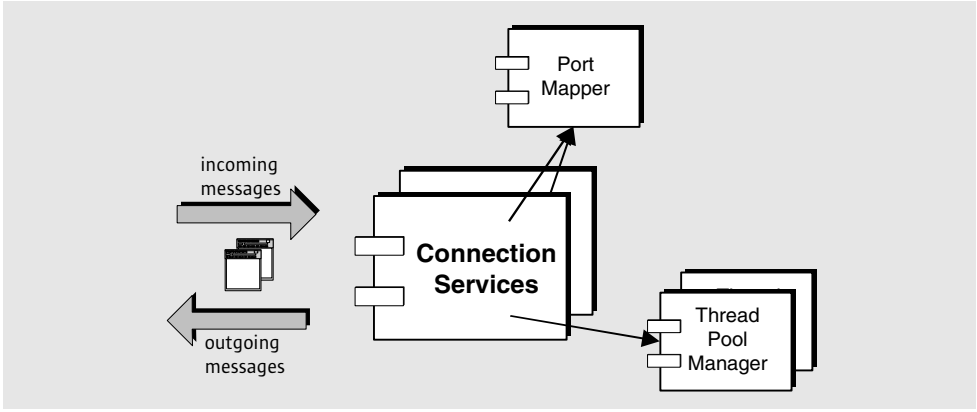
The connection services currently available from an iMQ broker are shown in Table 2-2:

Table 2-2 Connection Services Supported by an iMQ Broker

Service Name	Service Type	Connection Type (Transport Protocol)
jms	NORMAL (JMS message delivery)	tcp
ssljms	NORMAL (JMS message delivery)	ssl (over tcp)
httpjms	NORMAL (JMS message delivery)	http
admin	ADMIN	tcp

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

Figure 2-3 Connection Services Support



Each connection service is available at a particular port, specified by the broker's host name and a port number. The port can be statically or dynamically allocated. iMQ provides a *Port Mapper* that maps dynamically allocated ports to the different connection services. The Port Mapper itself resides at a standard port number, 7676. When a client sets up a connection with the broker, it first contacts the Port Mapper requesting the port number of the connection service it desires. You can also assign a *static* port number for each service when configuring broker connection services, but this is not recommended.

Each connection service is multi-threaded, supporting multiple connections. Each connection to the broker requires two threads: one to manage incoming messages and one to manage outgoing messages. To conserve resources, a *Thread Pool Manager* component allocates threads to connections, as needed, from a shared thread pool. You can configure the Thread Pool Manager to optimize connection resources. For example, you can set the minimum number and maximum number of threads in a thread pool.

Each connection service also supports specific authentication and authorization (access control) features (see "" on page 48).

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

Table 2-3 Connection Service Properties

Property Name	Description
`jmq.service.activelist`	List of connection services, by name, separated by commas, to be made active at broker startup. Supported services are: jms, ssljms, httpjms, admin. Default: `jms`, `admin`, `httpjms`
`jmq.`*service_name*`.min_threads`	The number of threads pre-allocated from the thread pool for the named connection service. Default: Depends on connection service (see Table 5-1 on page 106)
`jmq.`*service_name*`.max_threads`	The maximum number of threads allocated from the thread pool for the named connection service. The number must be greater than zero and greater in value than the value of `min_threads`. Default: Depends on connection service (see Table 5-1 on page 106)
`jmq.portmapper.port`	Port at which Port Mapper resides. If you are running more than one broker instance on a host, each must be assigned a unique Port Mapper port. Default: `7676`
`jmq.`*service_name*`.`*protocol_name*`.port` (protocol_name = connection type, as shown in Table 2-2)	Specifies the port number for the named connection service. Port number = 0 means the port is dynamically allocated by the Port Mapper. Default: `0`

Message Router

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

Basic Delivery Mechanisms

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

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

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

Once the Message Router has delivered a message to all its intended consumers it clears the message from memory, and if the message is persistent (see "Reliable Messaging" on page 33), removes it from the broker's persistent data store.

Reliable Delivery: Acknowledgements, and Transactions

The relatively straightforward delivery mechanism just described becomes more complicated when adding requirements for *reliable* delivery (see "Reliable Messaging" on page 33). There are two aspects involved in reliable delivery: assuring that delivery of messages to and from a broker is successful, and assuring that the broker does not lose messages or delivery information before messages are actually delivered.

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

For example, when a producer sends a JMS message (a payload message as opposed to a control message) to a destination, the broker sends back a control message—a broker acknowledgement—that it received the JMS message. (In practice, iMQ only does this if the producer specifies the JMS message as persistent.) The producing client uses the broker acknowledgement to guarantee delivery to the destination (see "Message Production" on page 64).

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

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

Furthermore, the Message Router confirms receipt of the client acknowledgement by sending a broker acknowledgement back to the client. The consuming client uses the broker acknowledgement to make sure that the broker will not deliver a JMS message more than once (see "Message Consumption" on page 65). This could happen if, for some reason, the broker fails to receive the client acknowledgement).

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

The client and broker acknowledgement processes described above apply, as well, to JMS message deliveries grouped into transactions. In such cases, client and broker acknowledgements operate on the level of a transaction rather than on the level of individual JMS message sends or receives. When a transaction commits, a broker acknowledgement is sent automatically. The broker employs a transaction manager to commit transactions or roll them back should they fail.

Reliable Delivery: Persistence

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

A producer client can specify that a message be persistent, and in this case, the Message Router will pass the message to a *Persistence Manager* that stores the message in a database or file system (see "Persistence Manager" on page 46) so that the message can be recovered if the broker fails.

Allocating Router System Resources

The performance of a broker depends on the system resources available and how efficiently resources such as memory are utilized. For example, the Message Router includes a mechanism for locally swapping messages to disk when memory resources become scarce.

You can configure the broker's memory management functions using properties that govern how often memory reclamation takes place, that set limits on the total number and total size of messages in memory, and that control the swapping of messages to disk. For example, you can set thresholds for the number of messages or the total size of messages that will trigger swapping (whichever is reached first will trigger the operation), and you can set the percentage of messages remaining after swapping takes place.

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

Table 2-4 Message Router Properties

Property Name	Description
`jmq.message.expiration.interval`	Specifies how often reclamation of expired messages occurs, in seconds. Default: `60`
`jmq.system.max_count`	Specifies maximum number of messages in both memory and disk (due to swapping). Additional messages will be rejected. A value of 0 means no limit. Default: `0`
`jmq.system.max_size`	Specifies maximum total size (in bytes, Kbytes, or Mbytes) of messages in both memory and disk (due to swapping). Additional messages will be rejected. A value of 0 means no limit. Default: `0`
`jmq.swap.threshold_count`	Swapping threshold: specifies the maximum number of in-process messages stored in memory before swapping to disk takes place. A value of 0 means no limit. Default: `0`
`jmq.swap.threshold_size`	Swapping threshold: specifies maximum size (in bytes, Kbytes, or Mbytes) of messages stored in memory before swapping to disk takes place. A value of 0 means no limit. Default: `70m`
`jmq.swap.percent`	Specifies the percentage of messages swapped to disk (percent by number or by size, depending on which triggered swapping). A value of 0 means no limit. Default: `50`
`jmq.message.max_size`	Specifies maximum allowed size (in bytes, Kbytes, or Mbytes) of a message body. Any message larger than this will be rejected. A value of 0 means no limit. Default: `70m`

Table 2-4 Message Router Properties *(Continued)*

Property Name	Description
`jmq.redelivered.optimization`	Specifies (true/false) whether Message Router optimizes performance by setting Redeliver flag whenever messages are re-delivered (true) or only when it is logically necessary to do so (false). Default: `true`

Persistence Manager

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

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

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

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

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

iMQ supports both built-in and plugged-in persistence modules. Built-in persistence is based on a flat file data store. Plugged-in persistence uses a JDBC interface and requires a JDBC-compliant data store. The built-in persistence is generally faster than plugged-in persistence; however, some users prefer the redundancy and administrative features of using a JDBC-compliant database system.

Built-in persistence

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

The flat file data store is located at:

```
JMQ_VARHOME/stores/brokerName/filestore/
```

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

Creating and deleting files, as data is added to and deleted from the data store, involves expensive file system operations. The iMQ implementation therefore reuses files: when a file is no longer needed, instead of being deleted, it is added to a pool of free files available for re-use. You can configure the size of this file pool. You can also specify the percentage of free files in the file pool that is tagged for re-use, as compared to being truncated (cleaned up) for re-use. The higher the percentage of truncated files, the more overhead is required to maintain the file pool, but the quicker the broker can be shut down (tagged files need to be cleaned up at shutdown).

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

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

Plugged-in persistence

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

Persistence-related configuration properties are detailed in Table 2-5 on page 48. (For instructions on setting these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 2-5 Persistence Properties

Property Name	Description
`jmq.persist.store`	Specifies whether the broker is using built-in, file-based (`file`) persistence or plugged-in JDBC-compliant (`jdbc`) persistence. Default: `file`
`jmq.persist.file.message.filepool.limit`	For built-in, file-based persistence, specifies the maximum number of free files available for reuse in the file pool. The larger the number the faster the broker can process persistent data. Free files in excess of this value will be deleted. The broker will create and delete additional files, in excess of this limit, as needed. Default: `10000`
`jmq.persist.file.message.filepool.cleanratio`	For built-in, file-based persistence, specifies the percentage of free files in the file pool that are kept in a *clean* state (truncated). The higher this value, the more overhead required to clean files during operation, however, the fewer files the broker needs to clean up at shutdown. Default: `60`
`jmq.persist.file.message.fdpool.limit`	For built-in, file-based persistence, specifies the maximum number of data files to keep open (that is, the size of the file descriptor pool). A larger number increases the performance of persistence operations, but at the expense of other broker operations that require file descriptors, such as creating client connections. Default: `25` (Solaris and Linux), `1024` (Windows)
`jmq.persist.file.sync.enabled`	Specifies whether persistence operations sycnchronize in-memory state with the physical storage device. If `true`, data loss due to system crash is eliminated, but at the expense of performance of persistence operations. Default: `false`

Security Manager

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

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

iMQ administrators populate an iMQ-provided user repository (see "Using a Flat-File User Repository" on page 162), or plug a pre-existing LDAP user repository into the iMQ Security Manager component.

Authentication

iMQ security supports password-based authentication. When a client requests a connection to a broker, the client must submit a user name and password. The iMQ Security Manager compares the name and password submitted by the client to those stored in the user repository. On transmitting the password from client to broker, the passwords are encoded using either base64 encoding or message digest, MD5. For more secure transmission, see "Encryption" on page 50. You can separately configure the type of encoding used by each connection service or set the encoding on a broker-wide basis.

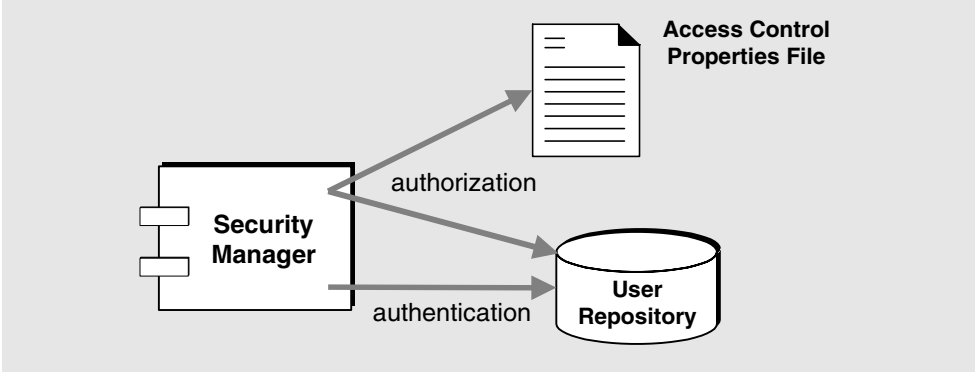
Authorization

Once the user of a client application has been authenticated, the user can be authorized to perform various iMQ-related activities. The iMQ Security Manager supports both user-based and group-based access control: depending on a user's name or the groups to which the user is assigned in the user repository, that user has permission to perform certain iMQ operations. You specify these access controls in an access control properties file (see Figure 2-4 on page 50).

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

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

Figure 2-4 Security Manager Support



For iMQ 2.0, the default access control properties file explicitly references only one group: *admin* (see "Groups" on page 164). A user in the *admin* group has admin service connection permission. The admin service lets the user perform administrative functions such as creating destinations, and monitoring and controlling a broker. A user in any other group you define cannot, by default, get an admin service connection.

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

In addition, if the broker is configured to allow auto-creation of destinations (see "Auto-Created (vs. Admin-Created) Destinations" on page 57), you can control for whom the broker can auto-create destinations by editing the access control properties file.

Encryption

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

To use the SSL connection service, you need to generate a private key/public key pair for the broker using the iMQ Key Tool utility (`jmqkeytool`). See "Encryption: Working With an SSL Service" on page 175.

The public key is embedded in a self-signed certificate before being placed in an iMQ keystore. The iMQ keystore is itself password protected; to unlock it, you have to provide a keystore password at broker startup time. Once the keystore is unlocked, the broker can pass the certificate to any client requesting a connection, and the client uses the certificate to set up an encrypted connection to the broker.

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

Table 2-6 Security Properties

Property Name	Description
`jmq.authentication.type`	Specifies whether password should be passed in base64 coding (`basic`) or as a MD5 digest (`digest`). Sets encoding for all connection services supported by a broker. Default: `digest`
`jmq.`*service_name*`.authentication.type`	Specifies whether password should be passed in base64 coding (`basic`) or as a MD5 digest (`digest`). Sets encoding for named connection service, overriding any broker-wide setting. Default: inherited from the value to which `jmq.authentication.type` is set.
`jmq.authentication.basic.user_repository`	Specifies (for base64 coding) the type of user repository used for authentication, either file-based (`file`) or LDAP (`ldap`). For additional LDAP properties, see Table 8-5 on page 168. Default: `file`
`jmq.authentication.client.response.timeout`	Specifies the time (in seconds) the system will wait for a client to respond to an authentication request from the broker. Default: `180 seconds`.
`jmq.accesscontrol.enabled`	Sets access control (`true/false`) for all connection services supported by a broker. Indicates whether system will check if an authenticated user has permission to use a connection service or to perform specific iMQ operations with respect to specific destinations, as specified in the access control properties file. Default: `true`.

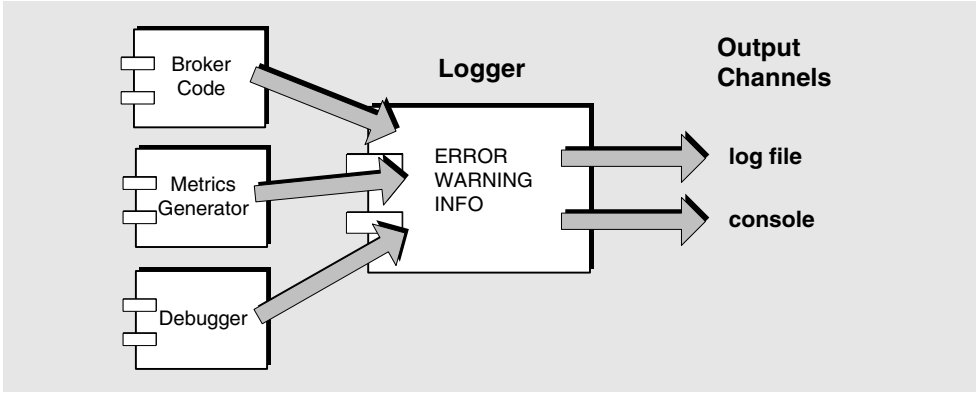
Table 2-6 Security Properties *(Continued)*

Property Name	Description
jmq.*service_name*.accesscontrol.enabled	Sets access control (true/false) for named connection service, overriding broker-wide setting. Indicates whether system will check if an authenticated user has permission to use the named connection service or to perform specific iMQ operations with respect to specific destinations, as specified in the access control properties file. Default: inherits the setting of the property jmq.accesscontrol.enabled.
jmq.accesscontrol.file.filename	Specifies the name of an access control properties file for all connection services supported by a broker. The file name specifies a relative file path to the directory JMQ_VARHOME/security. Default: accesscontrol.properties
jmq.*service_name*.accesscontrol.file.filename	Specifies the name of an access control properties file for named connection service. The file name specifies a relative file path to the directory JMQ_VARHOME/security. Default: inherits the setting specified by jmq.accesscontrol.file.filename
jmq.keystore.file.dirpath	For ssl-based services: specifies the path to the directory containing the keystore file. Default: JMQ_VARHOME/security
jmq.keystore.file.name	For ssl-based services: specifies the name of the keystore file. Default: jmqkeystore
jmq.keystore.password	For ssl-based services: specifies the keystore password so you are not prompted for it at broker startup. Default: NULL
jmq.keystore.passfile.enabled	For ssl-based services: specifies (true/false) if the keystore password is stored in a passfile. Default: false
jmq.keystore.passfile.dirpath	For ssl-based services: specifies the path to the directory containing the keystore passfile. Default: JMQ_VARHOME/security
jmq.keystore.passfile.name	For ssl-based services: specifies the name of the file containing the keystore password. Default: keypassfile

Logger

The broker includes a number of components for monitoring and diagnosing its operation. Among these are components that generate data (a Metrics Generator) and a Logger component that writes out data (metrics information as well as error messages and warnings) through a number of output channels. The scheme is illustrated in Figure 2-5.

Figure 2-5 Logging Scheme



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

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

Table 2-7 Logging Categories

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

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

The Logger can write data to two output channels: to standard output (the console) and to a log file. For each output channel you can specify which of the categories set to be logged by the Logger will be written to that channel. For example, you can specify that you want only errors and warnings written to the console, and only info (metrics) written to the log file.

In the case of a log file, you can also specify the point at which the log file is closed and output is rolled over to a new file. Once the log file reaches a specified size or age, it is saved and a new log file created. The log file is saved at the following location:

`JMQ_VARHOME/stores/`*brokerName*`/log/`

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

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

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

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

Table 2-8 Logger Properties

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

Table 2-8 Logger Properties *(Continued)*

Property Name	Description
`jmq.log.file.dirpath`	Specifies the path to the directory containing the log file. Default: `JMQ_VARHOME/stores/brokerName/log/`
`jmq.log.file.filename`	Specifies the name of the log file. Default: `log.txt`
`jmq.log.file.rolloverbytes`	Specifies the size, in bytes, of log file at which output rolls over to a new log file. A value of 0 means no rollover based on file size. Default: `0`
`jmq.log.file.rolloversecs`	Specifies the age, in seconds, of log file at which output rolls over to a new log file. A value of 0 means no rollover based on age of file. Default: `604800`
`jmq.log.console.output`	Specifies which categories of logging information are written to the console. Allowed values are: any set of logging categories separated by vertical bars (`|`), or `ALL`, or `NONE`. Default: `ERROR| WARNING`
`jmq.log.console.stream`	Specifies whether console output is written to stdout (`OUT`) or stderr (`ERR`). Default: `ERR`

Physical Destinations

iMQ messaging is premised on a two-phase delivery of messages: first, delivery of a message from a producer client to a destination on the broker, and second, delivery of the message from the destination on the broker to one or more consumer clients. There are two types of destinations (see "Programming Domains" on page 30): queues (point to point delivery model) and topics (publish/subscribe delivery model). These destinations represent locations in a broker's physical memory where incoming messages are marshaled before being routed to consumer clients.

You create physical destinations using iMQ administration tools (see "Managing Destinations" on page 137). Destinations can also be automatically created as described in "Auto-Created (vs. Admin-Created) Destinations" on page 57.

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

Queue Destinations

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

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

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

Since messages can remain in a queue for an extended period of time, memory resources can become an issue. You don't want to allocate too much memory to a queue (memory is under utilized), nor do you want to allocate too little (messages will be rejected). To allow for flexibility, based on the load demands of each queue, you can set physical properties when creating a queue: maximum number of messages in queue, maximum memory allocated for messages in queue, and maximum size of any message in queue (see Table 6-9 on page 139).

Topic Destinations

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

Messages do not normally remain in a topic destination for an extended period of time, so memory resources are not normally a big issue. However, you can configure the maximum size allowed for any message received by the destination (see Table 6-9 on page 139).

Auto-Created (vs. Admin-Created) Destinations

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

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

iMQ supports this *auto-create* capability. When auto-creation is enabled, a broker automatically creates a destination whenever a MessageConsumer or MessageProducer attempts to access a non-existent destination. (The user of the client application must have auto-create privileges—see "Destination Auto-Create Access Control" on page 175).

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

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

Table 2-9 Auto-create Configuration Properties

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

Temporary Destinations

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

Multi-Broker Configurations (Clusters)

iMQ supports the implementation of a Message Service using multiple interconnected brokers—a broker cluster. Cluster support provides for scalability of your Message Service.

As the number of clients connected to a broker increases, and as the number of messages being delivered increases, a broker will eventually exceed its resource limitations: for example, file descriptor and memory limits. One way to accommodate increasing loads is to add more brokers to an iMQ Message Service, distributing client connections and message delivery across multiple brokers.

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

While there are other reasons for using broker clusters (for example, to accommodate workgroups having different user repositories, or to deal with firewall restrictions), failover is *not* one of them. One broker in a cluster cannot be used as an automatic backup for another that fails. Automatic failover protection for brokers is not supported in iMQ 2.0. (However, an application could be designed to use multiple brokers to implement a customized failover scheme.)

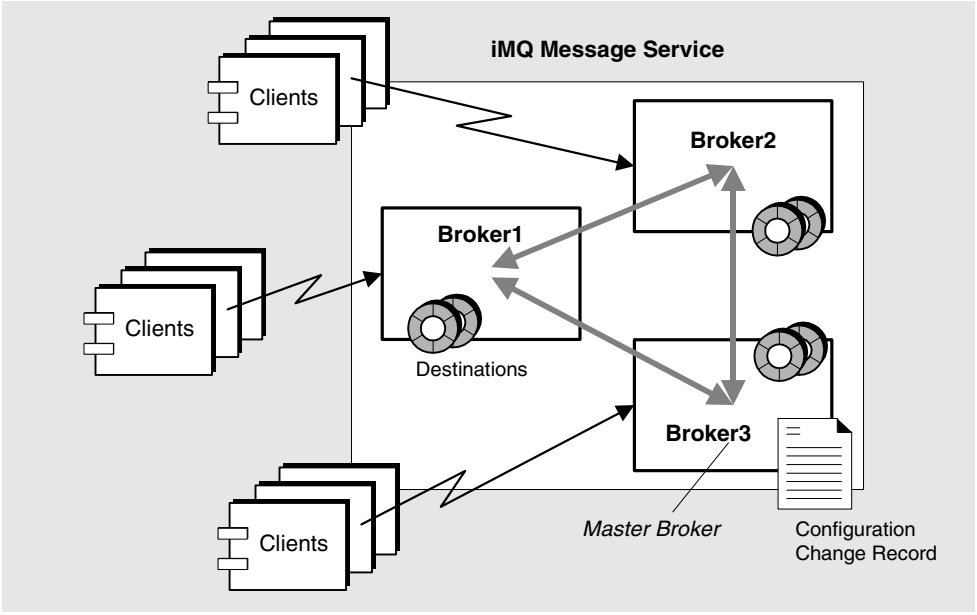
Information on configuring and managing a broker cluster is provided in "Working With Broker Clusters" on page 113. The following sections explain the architecture and internal functioning of iMQ broker clusters.

Multi-Broker Architecture

A multi-broker Message Service allows client connections to be distributed among a number of brokers, as shown in Figure 2-6 on page 60. From a client point of view, each client connects to an individual broker (its *home* broker) and sends and receives messages as if the home broker were the only broker in the cluster. However, from a Message Service point of view, the home broker is working in tandem with other brokers in the cluster to provide delivery services to the message producers and consumers to which it is directly connected.

In general, the brokers within a cluster can be connected in any arbitrary topology. However, iMQ 2.0 only supports fully-connected clusters, that is, a topology in which every broker is directly connected to every other broker in the cluster, as shown in Figure 2-6.

Figure 2-6 Cluster Architecture



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

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

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

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

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

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

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

In a mission-critical application environment it is a good idea to make a periodic backup of the configuration change record to guard against accidental corruption of the record and safeguard against Master Broker failure. You can do this using the `-backup` option of the `jmqbroker` command (see Table 5-2 on page 110), which provides a way to create a backup file containing the configuration change record. You can subsequently restore the configuration change record using the `-restore` option.

If necessary you can change the broker serving as the Master Broker by backing up the configuration change record, modifying the appropriate cluster configuration property (see Table 2-10 on page 63) to designate a new Master Broker, and restarting the new Master Broker using the `-restore` option.

Using Clusters in Development Environments

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

Under test situations, destinations are generally auto-created (see "Auto-Created (vs. Admin-Created) Destinations" on page 57) and durable subscriptions to these destinations are created and destroyed by the applications being tested. These changes in destinations and durable subscriptions will be propagated throughout the cluster. However, if you reconfigure the environment to use a Master Broker, iMQ will re-impose the requirement that the Master Broker be running for changes to be made in destinations and durable subscriptions, and for these changes to be propagated throughout the cluster.

Cluster Configuration Properties

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

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

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

iMQ cluster configuration properties are shown in Table 2-10. (For instructions on setting these properties, see "Working With Broker Clusters" on page 113.)

Table 2-10 Cluster Configuration Properties

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

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

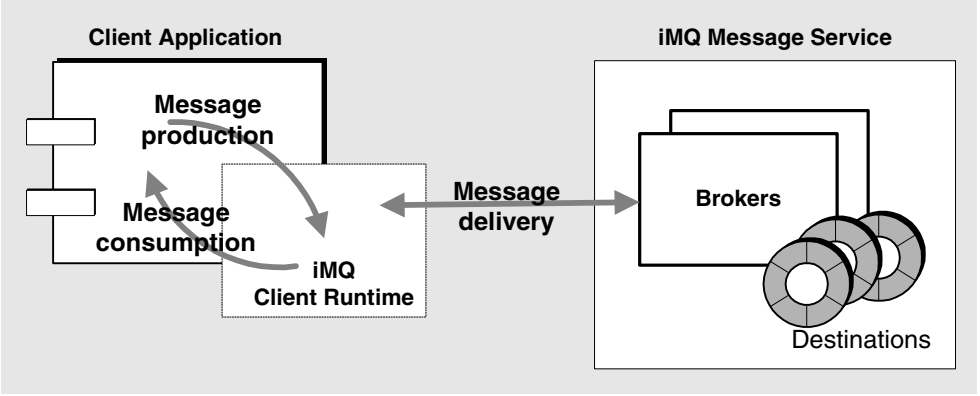
iMQ Client Runtime

The iMQ client runtime provides client applications with an interface to the iMQ Message Service—it supplies client applications with all the JMS programming objects introduced in "The JMS Programming Model" on page 27. It supports all operations needed for clients to send messages to destinations and to receive messages from such destinations.

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

Figure 2-7 on page 64 illustrates how message production and consumption involve an interaction between client applications and the iMQ client runtime, while message delivery involves an interaction between the iMQ client runtime and the iMQ Message Service.

Figure 2-7 Messaging Operations



Message Production

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

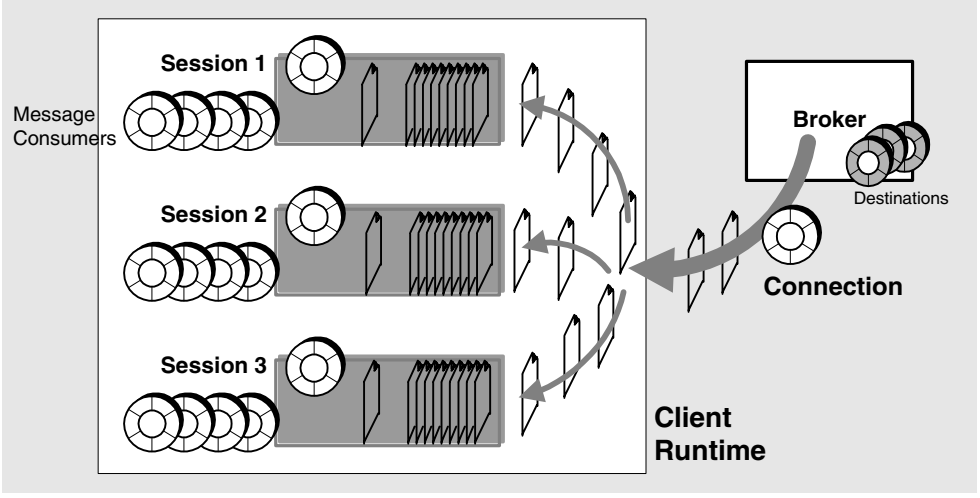
Message Consumption

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

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

Messages delivered over the connection are distributed to the appropriate iMQ sessions where they are queued up to be consumed by the appropriate MessageConsumer objects, as shown in Figure 2-8. Messages are fetched off each session queue one at a time (a session is single threaded) and consumed either synchronously (by a client thread invoking the `receive` method) or asynchronously (by the session thread invoking the `onMessage` method of a MessageListener object).

Figure 2-8 Message Delivery to Client Runtime



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

iMQ Administered Objects

Administered Objects encapsulate provider-specific implementation and configuration information in objects that are used by client applications. Administered objects are created and configured by an administrator, stored in a name service, accessed by client applications through standard JNDI lookup code, and then used in a provider-independent manner.

iMQ provides two types of administered objects: `ConnectionFactory` and `Destination`. While both encapsulate provider-specific information, they have very different uses within a client application. `ConnectionFactory` objects are used to create connections to the Message Service and `Destination` objects (which represent physical destinations) are used to identify physical destinations.

Administered objects make it very easy to control and manage an iMQ Message Service:

- You can control the behavior of connections by requiring client applications to access pre-configured `ConnectionFactory` objects through a JNDI lookup.
- You can control the proliferation of physical destinations by requiring client applications to access only `Destination` objects that correspond to existing physical destinations. (You also have to disable the brokers's auto-create capability—see "Auto-Created (vs. Admin-Created) Destinations" on page 57).

This arrangement therefore gives you control over Message Service configuration details, and at the same time allows client applications to be provider-independent: they do not have to know about provider-specific syntax and object naming conventions (see "JMS Provider Independence" on page 31) or provider-specific configuration properties.

You create administered objects using iMQ administration tools, as described in Chapter 7, "Managing Administered Objects". When creating an administered object, you can specify that it be read only—that is, client applications are prevented from changing iMQ-specific configuration values you have set when creating the object. In other words, client code cannot set attribute values on read-only administered objects, nor can you override these values using client application startup options, as described in "Starting Client Applications With Overrides" on page 68.

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

Connection Factory Administered Objects

A `ConnectionFactory` object is used to create physical connections between a client application and an iMQ Message Service. A `ConnectionFactory` object has no physical representation in a broker—it is used simply to enable the client application to establish connections with a broker. A `ConnectionFactory` object is also used to specify behaviors of the connection and of the client runtime that is using it to access a broker. By configuring a `ConnectionFactory` administered object, you specify the attribute values (the properties) common to all the connections that it produces.

To create a `ConnectionFactory` administered object, see "Adding a Connection Factory" on page 154.

`ConnectionFactory` attributes, are grouped into a number of categories, depending on the behaviors they affect:

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

Each of these categories and its corresponding attributes is discussed in some detail in the *iMQ Developer's Guide*. While you, as an iMQ administrator, might be called upon to adjust the values of these attributes, it is normally an application developer who decides which attributes need adjustment to tune the performance of client applications. Table Table 7-3 on page 147 presents an alphabetical summary of the attributes.

Destination Administered Objects

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

To create a `Destintion` administered object, see "Adding a Topic or Queue" on page 155.

Table 2-11 Destination Attributes

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

Starting Client Applications With Overrides

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

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

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

where `attribute` corresponds to any of the `ConnectionFactory` administered object attributes documented in "Connection Factory Objects" on page 147.

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

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

Chapter 3

iMQ Administration

iMQ administration consists of a number of tasks and a number of tools for performing those tasks.

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

iMQ Administration Tasks

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

Development Environments

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

Production Environments

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

Setup Operations

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

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

    (At a minimum, you want to password protect administration capability.)
  - modify access settings in the access control properties file
  - set up SSL-based connection services
- administered objects (see Chapter 7, "Managing Administered Objects"):
  - configure or set up an LDAP object store
  - create ConnectionFactory and Destination administered objects
- broker clusters (see "Working With Broker Clusters" on page 113):
  - create a central configuration file
  - use a Master Broker
- persistence: configure the broker to use plugged-in persistence, rather than built-in persistence (see Appendix A, "Setting Up Plugged-in Persistence")

Maintenance Operations

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

- application management:
  - disable the broker's auto-create capability (see Table 2-9 on page 58)
  - create physical destinations on behalf of applications (see "Creating Destinations" on page 138)
  - set user access to destinations (see "Authorizing Users: the Access Control Properties File" on page 170)
  - monitor and manage destinations (see "Managing Destinations" on page 137)
  - monitor and manage durable subscribers (see "Managing Durable Subscriptions" on page 142)
- broker administration and tuning:
  - use broker metrics to control and tune the broker
  - manage broker memory resources
  - add brokers to clusters to balance loads
  - recover failed brokers
- managing administered objects
  - adjust ConnectionFactory attribute values to improve performance and throughput (see "Connection Factory Administered Objects" on page 67).

iMQ Administration Tools

iMQ administration tools fall into two categories: command line utilities and a graphical user interface (GUI) Administration Console. You can use the Administration Console to manage a broker remotely and to manage iMQ administered objects. The Console combines the capabilities of two command line utilities (JMQ Command and JMQ Object Manager), which you can also use remotely. The other command line utilities must be run on the same host as their associated broker, as shown in Figure 3-1.

Most of the command line utilities are used to perform specialized tasks (see "Summary of Command Line Utilities" on page 72). Information on the iMQ Administration Console, is available in the online help.

The Administration Console

You can use the administration console to do the following:

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

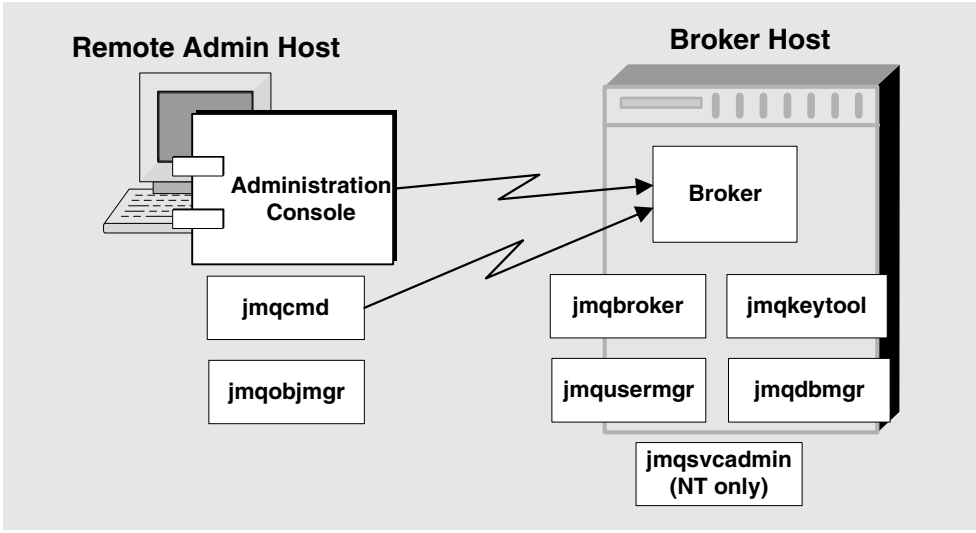
There are some tasks that you cannot use the Administration Console to complete; chief among these are configuring the broker, creating broker clusters, and managing the user database.

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

Summary of Command Line Utilities

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

Figure 3-1 Local and Remote Utilities



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

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

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

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

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

iMQ Key Tool (`jmqkeytool`) You use the iMQ Key Tool utility to generate a self-signed certificate used for SSL authentication and to place it in iMQ's keystore file. You use the `jmqkeytool` command to run this utility, which is described in Chapter 8, "Security Management."

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

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

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

Command Line Syntax

All the command line utilities share the following command syntax:

Utility_Name [*Subcommand_Clause*]

Utility_Name specifies the name of an iMQ utility, for example, `jmqcmd`, `jmqobjmgr`, `jmqusermgr`, and so on.

Subcommand_Clause, which is optional for some commands, has the following syntax

[*Subcommand*] [*target*] [[*-option_name* [*operand*]]...]

There are four important things to remember:

- Specify options *after* subcommands (and targets) if the utility accepts both types of arguments.
- If an argument contains a space, enclose the whole argument in quotation marks. It is generally safest to enclose an attribute-value pair in quotes.

- If you specify the -v (version) or the -h/-H (help) options on a command line, nothing else on that command line is executed. See Table 3-1 on page 75 for a description of common options.
- Separate subcommand arguments (target, option name, operands) with spaces.

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

```
jmqbroker
```

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

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

Common Utility Options

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

Table 3-1 Common iMQ Utility Options

Option	Description
`-h`	Displays usage help for the specified utility.
`-H`	(Not universal.) Displays expanded usage help, including attribute list and examples.
`-s`	Turns on silent mode: no output is displayed. Specify as `-silent` for `jmqbroker`.
`-v`	Displays version information.
`-f`	Performs the given action without prompting for user confirmation.
`-pre`	(Used only with `jmqobjmgr`) Turns on preview mode, allowing the user to see the effect of the rest of the command line without actually performing the command. This can be useful in checking for the value of default attributes.
`-javahome` *path*	Specifies the location of an alternate Java 2 runtime to use.

System Resources: Windows 98

If you encounter an "Out of environment space" error while running any of the administration tools, you must increase the size of the environment table. You can do this from the command line, to temporarily increase the table size, or you can edit a configuration file, to permanently increase the table size.

➤ **To temporarily increase the Windows environment table size**

1. Enter the following command at the DOS prompt:

   ```
   command /e:8192
   ```

2. Resume your work.

➤ **To permanently increase the Windows environment table size**

1. Add the following line to the `CONFIG.SYS` file.

   ```
   shell=command.com /e:8192 /p
   ```

2. Restart your computer.

iMQ Administration Console Tutorial

This tutorial focuses on the use of the iMQ Administration Console, a graphical interface for administering an iMQ Message Service. By following this tutorial, you will learn how to do the following:

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

Finally, the tutorial will demonstrate a working broker by having you run the JMS-compliant application `SimpleAdmin`.

This tutorial is provided mainly to guide you through performing basic administrative tasks using the Administration Console. It is not a substitute for reading through the *iMQ Developer's Guide* or the *iMQ Administrator's Guide*.

Note also that some tasks cannot be accomplished using graphical tools; you will need to use the command line utilities as well. Chief among these tasks are the following:

- Configuring the broker

  See Chapter 5, "Starting and Configuring a Broker" on page 103 for more information.

- Creating broker clusters

  See"Working With Broker Clusters" on page 113 for more information.

- Managing the user database

  See "Authenticating Users" on page 162 for more information.

Getting Ready

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

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

Starting the Administration Console

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

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

  Administered objects allow you to administer the messaging needs of JMS-compliant applications. For more information, see "iMQ Administered Objects" on page 66.

➤ **To start the Administration Console**

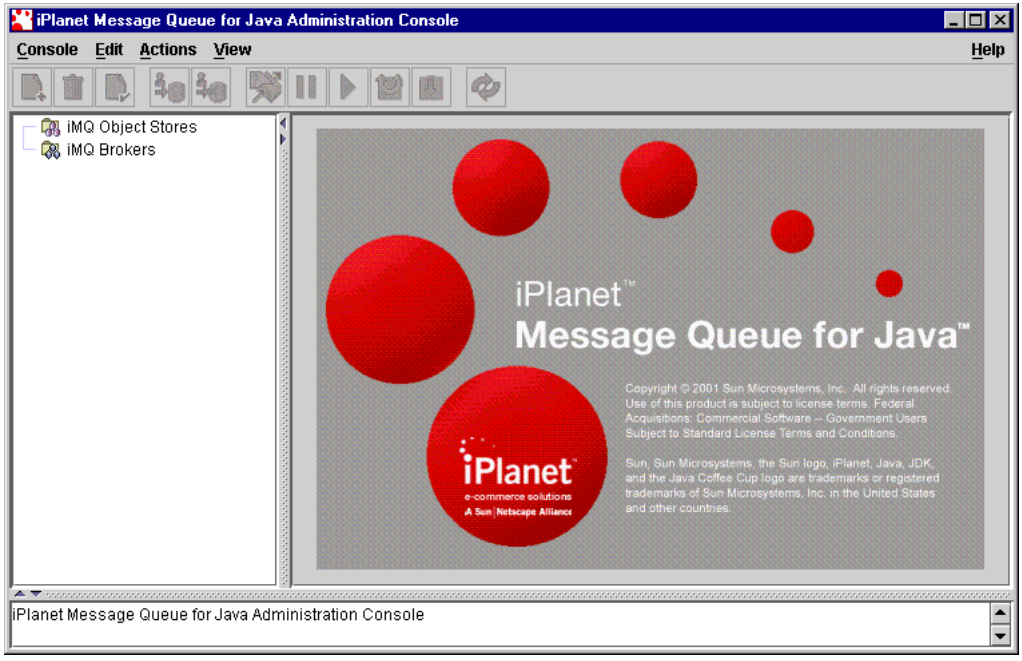
1. Choose Start > Programs > iPlanet Message Queue 2.0 > iMQ Administration.

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

   Linux/unix users: enter the following command at the command prompt:

   ```
   $JMQ_HOME/bin/jmqadmin
   ```

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



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

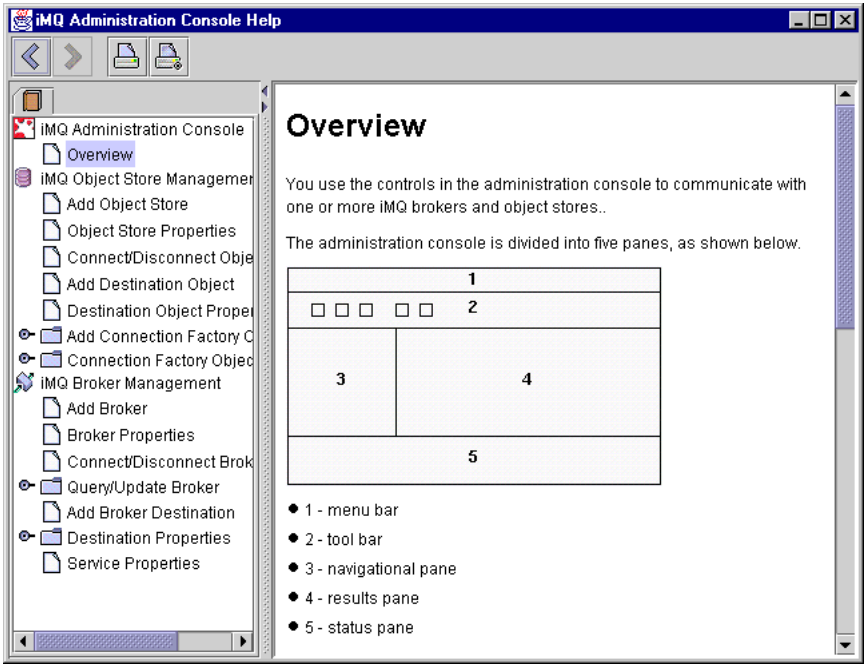
No tutorial can provide complete information, so let's first find out how to get help information for the Administration Console.

Getting Help

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

➤ **To display Administration Console help information**

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



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

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

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

   Your next task, will be to add a broker. Before you start, check the online help for information.

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

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

4. Read through the help text.

5. Close the Help window.

Working With Brokers

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

Working with brokers involves the following tasks:

- Start and configure the broker

  You can start the broker from the Start > Programs menu on the NT or by using the jmqbroker utility. If you use the utility, you can specify configuration information. If you use the Programs menu, you need to specify configuration information in other ways. See Chapter 5, "Starting and Configuring a Broker" for more information.

- Manage the broker and its services either by using the Administration Console or by using the jmqcmd utility

- Create the physical destinations needed by client applications

- Monitor resource use to improve throughput and reliability

The broker supports communication with both application clients and administration clients. It does this by means of different connection services, and you can configure the broker to run any or all of these services. For more information about connection services, see "Connection Services" on page 40.

Starting a Broker

Currently, you cannot start a broker using the Administration Console.

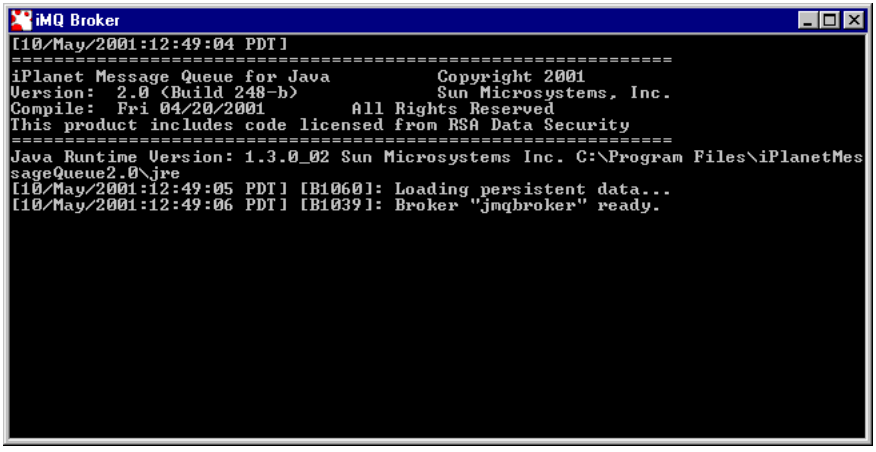
➤ **To start a broker**

1. Choose Start > Programs > iPlanet Message Queue 2.0 > iMQBroker.

   Solaris/Linux: enter the following command to start a broker.

   ```
   %$JMQ_HOME/bin/jmqbroker
   ```

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



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

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

Adding a Broker

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

➤ **To add a broker to the Administration Console**

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

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

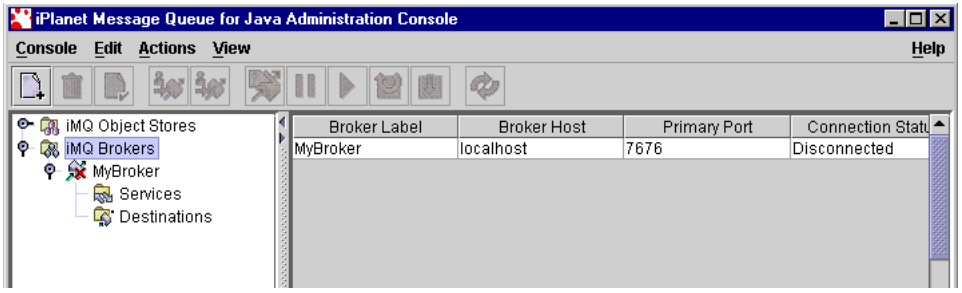
   This provides a label that identifies the broker in the Administration Console.

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

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

3. Click OK to add the broker.

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



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

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

Changing the Administrator Password

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

➤ **To change the administrator password**

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

   ```
   jmqusermgr update -u admin -p abracadabra
   ```

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

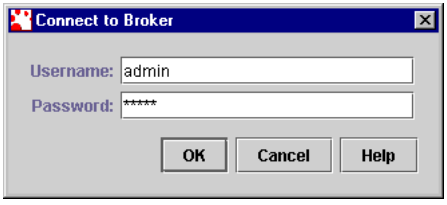
Although clients use a different iMQ service than administrators, they are also assigned a default user name and password so that you can test iMQ without having to do extensive administrative set up. By default, a client can connect to the broker as user `guest` with the password `guest`. You should, however, establish secure user names and passwords for clients as soon as you can. See "Authenticating Users" on page 162 for more information.

Connecting to the Broker

➤ **To connect to the broker**

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

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



2. Enter `admin` in the Password field or whatever value you specified for the password in "Changing the Administrator Password" on page 84.

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

3. Click OK to connect to the broker.

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

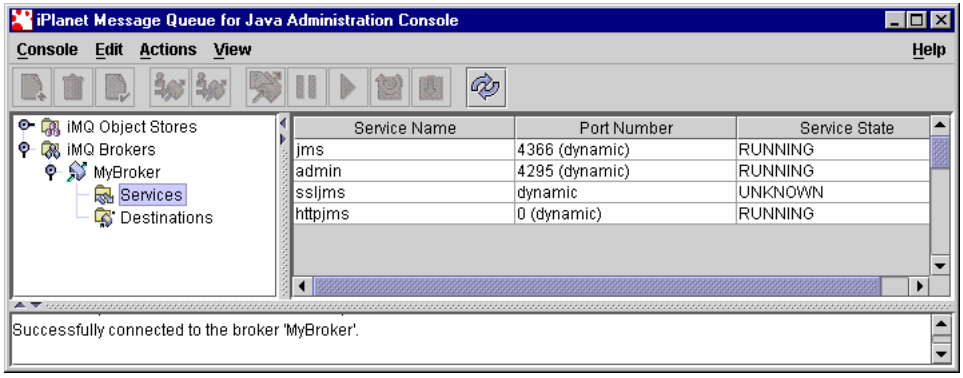
Broker Service Properties

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

➤ **To view available connection services**

1. Select Services in the navigation pane.

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



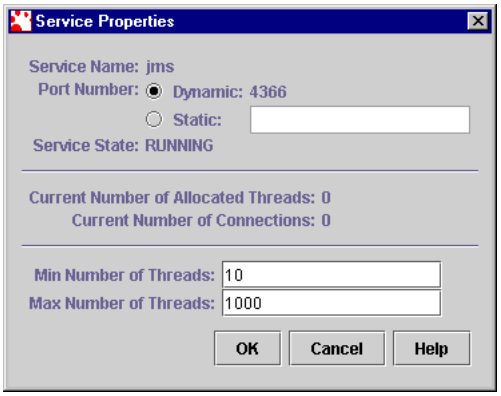
2. Select the jms service by clicking on it in the results pane.

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

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

4. Choose Properties from the Actions menu.

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



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

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

Adding Physical Destinations to a Broker

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

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

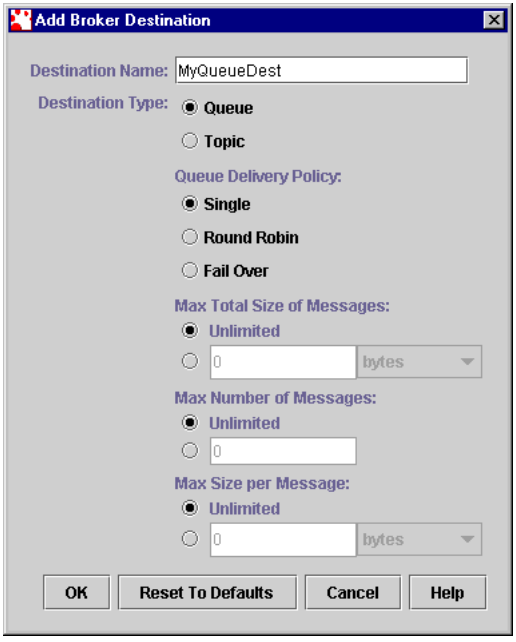
You control whether the broker can add auto-created destinations by setting the `jmq.autocreate.topic` or `jmq.autocreate.queue` properties. For more information, see "Auto-Created (vs. Admin-Created) Destinations" on page 57.

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

➤ To add a queue destination to a broker

1. Right-click the Destinations subdirectory of iMQBrokers and choose Add Broker Destination.

   The following dialog is displayed:



2. Enter `MyQueueDest` in the Destination Name field.
3. Select the Queue radio button if it is not already selected.
4. Make sure the Queue Delivery Policy is selected as Single.
5. Click OK to add the physical destination.

Working With Physical Destinations

Once you have added a physical destination on the broker, you can do any of the following:

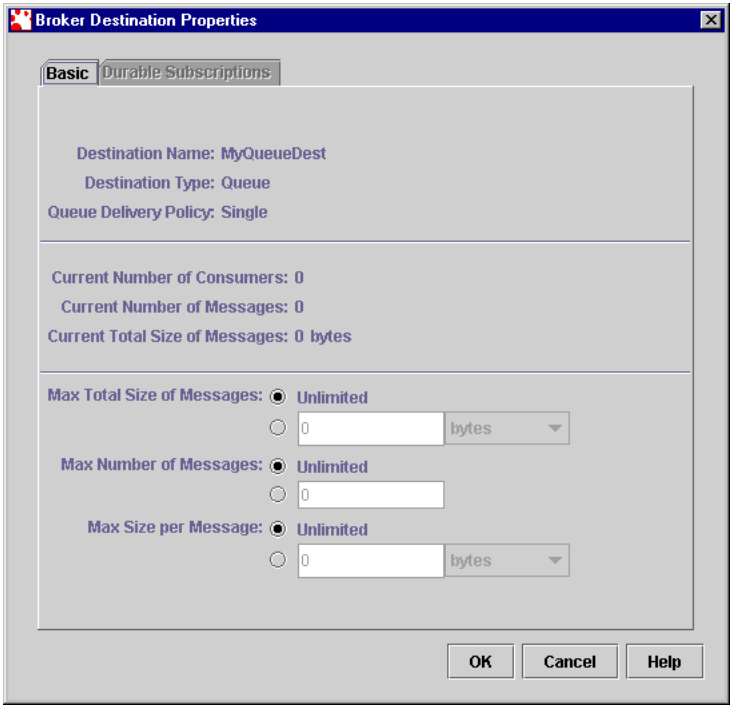
- View and update the properties of a physical destination
- Purge messages at a destination
- Delete a destination

The following procedures explain how to do each of these tasks.

➤ **To view the properties of a physical destination**

1. Select MyQueueDest in the results pane.
2. Choose Actions > Properties.

   The following dialog is displayed:



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

3. Click Cancel to close the dialog.

➤ **To purge messages from a destination**

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

   A confirmation dialog is displayed.

Purging messages removes the messages and leaves an empty destination.

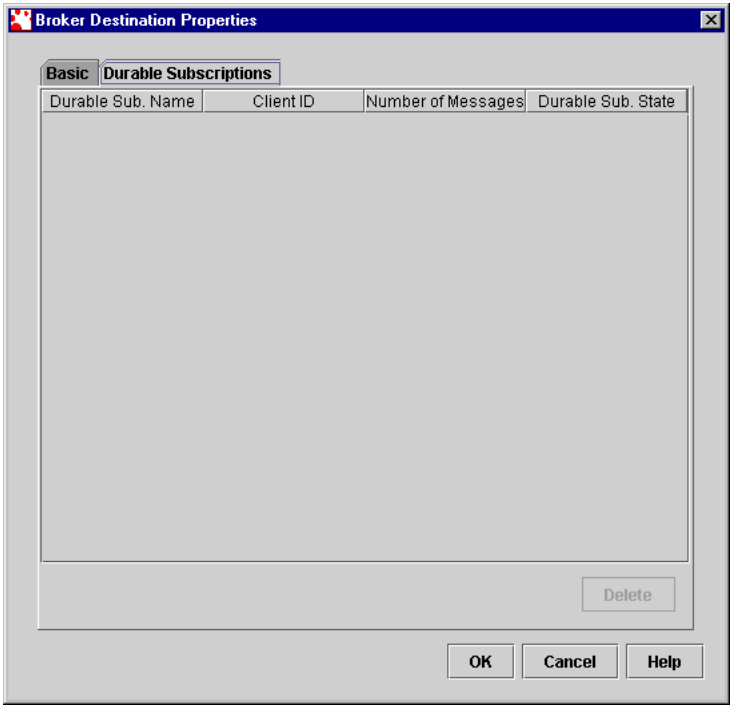
➤ **To delete a destination**

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

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

Getting Information About Topic Destinations

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



You can use this dialog to delete durable subscriptions for any subscriber that no longer exists.

Working with Object Stores

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

Although administered objects can be instantiated and configured within client code, the approach that actually allows messaging systems to be centrally administered and provider-independent involves you, the administrator, creating and configuring these objects and storing them in an object store that is accessed by client applications through standard JNDI lookup code.

For more information about administered objects, see "iMQ Administered Objects" on page 66.

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

Adding an Object Store

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

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

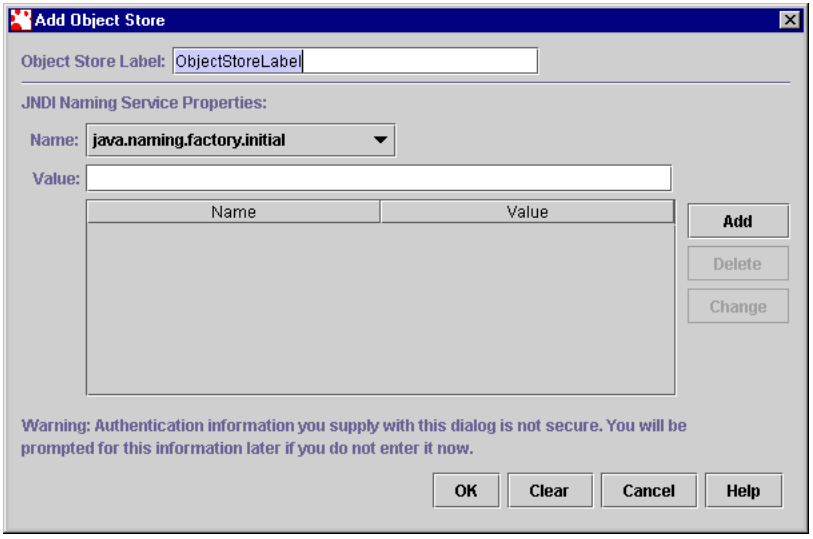
1. If you do not already have a folder named `Temp` on your C drive, create it now.

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

   Solaris/linux users: you can use the `/tmp` directory which should already exist.

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

   The following dialog is displayed:



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

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

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

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

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

5. In the Value field, enter the following

   `com.sun.jndi.fscontext.RefFSContextFactory`

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

   In a production environment, you will probably want to use an LDAP directory server as an object store. For information about setting up the server and doing JNDI lookups, see "Object Store Attributes" on page 149.

6. Click the Add button.

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

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

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

8. In the Value field, enter the following

   `file:///C:/Temp`

   If you are working in unix, specify `file:/tmp`

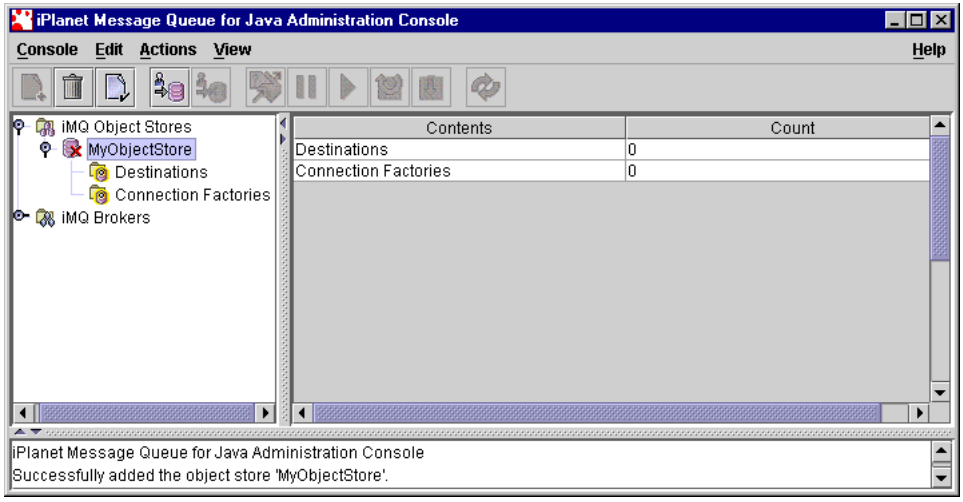
9. Click the Add button.

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

10. Click OK to add the object store.

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

    The Administration Console now looks like this:



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

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

Checking Object Store Properties

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

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

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

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

Connecting to an Object Store

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

➤ **To connect to an object store**

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

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

Adding and Configuring a Connection Factory Object

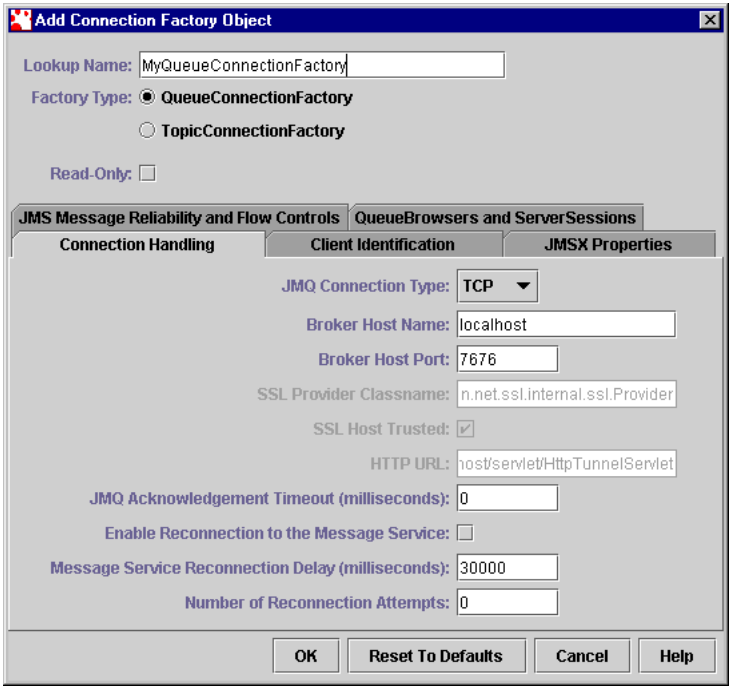
You can use the administration console to create and configure a connection factory. A connection factory is used by client code to connect to the broker. By configuring a connection factory, you can control the behavior of the connections it is used to create.

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

➤ **To add a connection factory to an object store**

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

   The Add Connection Factory Object dialog is displayed.



2. Enter the name "`MyQueueConnectionFactory`" in the LookupName field.

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

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

3. Select the QueueConnectionFactory radio button to specify the type of the connection factory.

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

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

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

6. Click OK to create the queue connection factory.

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

Adding a Destination Object

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

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

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

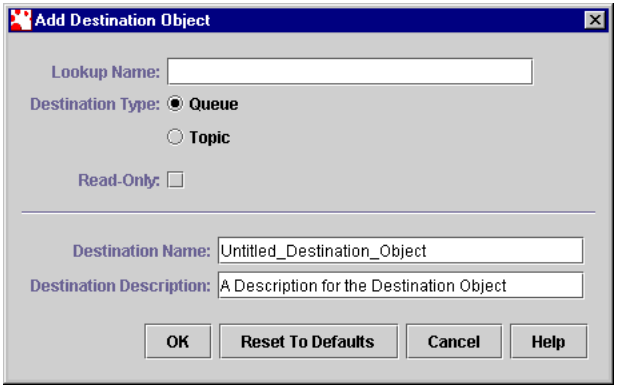
- If you have not created a physical destination and the autocreation of physical destinations is *disabled*, the broker cannot create a physical destination and cannot deliver the message.

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

➤ **To add a destination to an object store**

1. Right-click on the Destinations node in the navigation pane.
2. Choose Add Destination Object.

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



3. Enter "`MyQueue`" in the Lookup Name field.

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

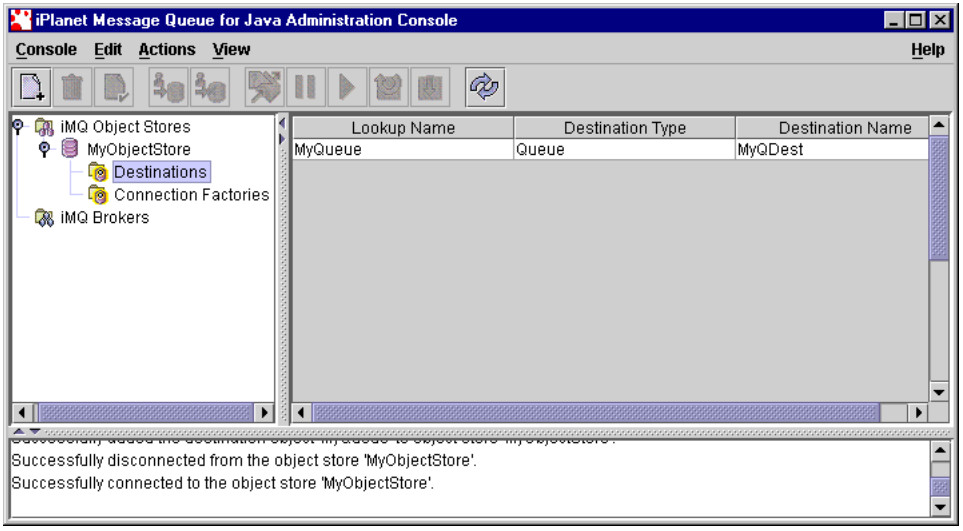
   ```
   queue=(javax.jms.Queue)ctx.lookup("MyQueue");
   ```

4. Select the Queue radio button for the Destination Type.
5. Enter `MyQueueDest` in the Destination Name field.

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

6. Click OK.

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



Administered Object Properties

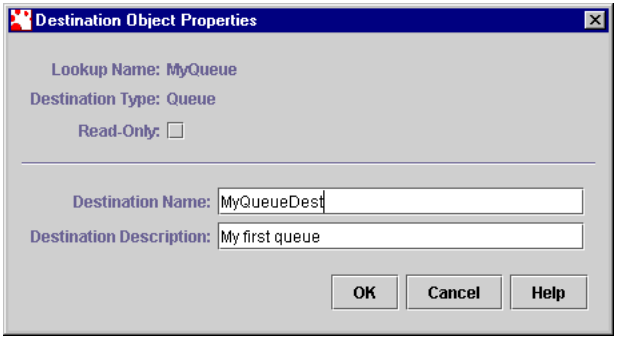
To view or update the properties of an administrative object, you need to select Destinations or Connection Factories in the navigation pane, select a specific object in the results pane, and choose Actions > Properties.

➤ **To view or update the properties of a destination object**

1. Select Destinations in the navigation pane.
2. Select MyQueue in the results pane.

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

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



Updating Console Information

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

Running the Sample Application

The sample application `SimpleAdmin.java` is provided for use with this tutorial. The code creates a simple queue sender and receiver. The sample code relies on the administrator having created a queue connection factory named `MyQueueConnectionFactory` and a queue named `MyQueue`. This is what you have done in the tutorial so far.

➤ **To download the SimpleAdmin.java application**

1. Go to the following url:

   `http://docs.iplanet.com/docs/manuals/javamq.html`

2. In the iPlanet Message Queue for Java 2.0 pane, look for the topic Administration Guide.

3. Click on Sample Code to download the source file `SimpleAdmin.java`.

Before you run the application, open the source file and read through the source. It is short, but it is amply documented and it should be fairly clear how it uses the objects and destinations we have created using the tutorial.

➤ **To prepare for compiling and running the sample application**

1. Make sure that the `JMQ_HOME` environment variable is set.

2. Set the `JAVA_HOME` environment variable to the directory where you installed the J2SE SDK.

3. Make the directory that includes the `SimpleAdmin.java` file your current directory; for example

   ```
   cd JMQ_HOME/examples/jms
   ```

4. Set the `CLASSPATH` variable to include the current directory containing `SimpleAdmin.java` as well as the following jar files: `jms.jar`, `jmq.jar`, `jndi.jar`, `fscontext.jar`, `providerutil.jar`.

➤ **To compile and run the sample application**

1. Set the `Path` variable to include the location of the `javac` compiler.
2. Compile the `SimpleAdmin.java` file as shown below:

   ```
   C:\> javac SimpleAdmin.java
   ```

   This results in the `SimpleAdmin.class` file being created in the current directory.

   On **Solaris/Linux** enter the following command:

   ```
   $JAVA_HOME/bin/javac SimpleAdmin.java
   ```

3. Set the `CLASSPATH` variable to include the location of the `SimpleAdmin.class` file that was output by the previous compilation.
4. Run the `SimpleAdmin` application.

   ```
   C:\> java SimpleAdmin
   ```

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

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

   On **Solaris/Linux** enter the following command:

   ```
   $JAVA_HOME/bin/java SimpleAdmin
   ```

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

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

Chapter 5

Starting and Configuring a Broker

After installing iMQ, you use the `jmqbroker` command to start a broker. The configuration of the broker is governed by a set of configuration files and by options passed with the `jmqbroker` command, which override corresponding properties in the configuration files.

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

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

For a description of how to start and use the broker as a Windows service, see "Using a Broker as a Windows Service" on page 191.

Configuration Files

Installed configuration files, which are used to configure the broker, are located in the `JMQ_HOME/props/broker/` directory. This directory stores the following files:

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

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

```
JMQ_VARHOME/stores/brokerName/props/config.properties
```

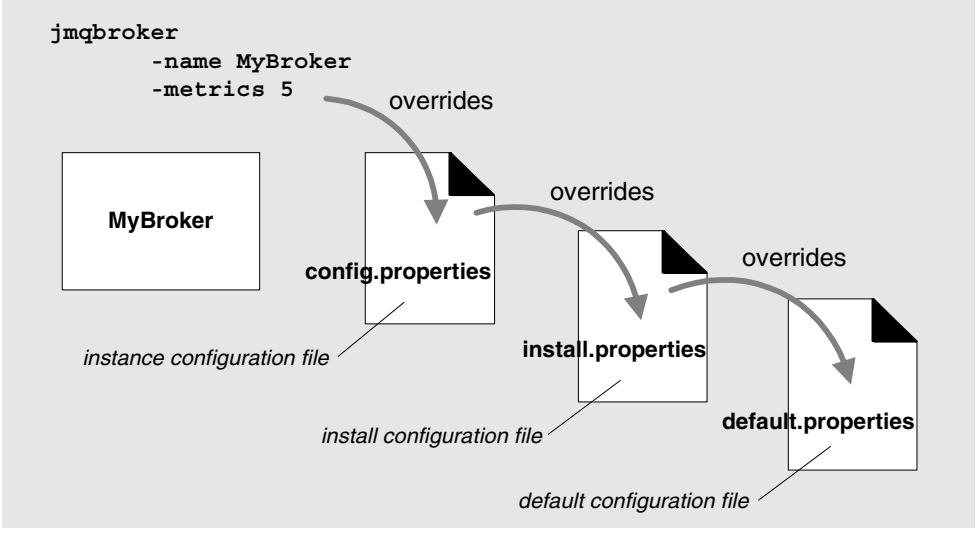
Where *brokerName* is the instance name of the broker (`jmqbroker` by default). You can edit an instance configuration file to make configuration changes (see "Editing the Instance Configuration File" on page 105).

If you connect brokers in a cluster (see "Multi-Broker Configurations (Clusters)" on page 58) you may also need to use a *cluster configuration file* to specify cluster configuration information. For more information, see "Cluster Configuration Properties" on page 113.

Merging Property Values

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

Figure 5-1 Configuration Files



Property Naming Syntax

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

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

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

```
jmq.queue.default=single
```

The following entry defines the message expiration timeout value:

```
jmq.message.expiration.timeout=90
```

Table 5-1 on page 106 lists the broker configuration properties (and their default values) in alphabetical order.

Editing the Instance Configuration File

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

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

- use administration tools. For information about properties you can set using `jmqcmd`, see Table 6-4 on page 132.
- edit the `config.properties` file while the broker is shut down; then restart the broker.

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

Table 5-1 Broker Configuration Properties

Property Name	Type	Default Value	Reference
`jmq.accesscontrol.enabled`	boolean	`true`	Table 2-6 on page 51
`jmq.accesscontrol.file.filename`	string	`accesscontrol.properties`	Table 2-6 on page 51
`jmq.authentication.basic.user_repository`	string	`file`	Table 2-6 on page 51
`jmq.authentication.client.response.timeout`	integer (seconds)	`180`	Table 2-6 on page 51
`jmq.authentication.type`	string	`digest`	Table 2-6 on page 51
`jmq.autocreate.queue`	boolean	`false`	Table 2-9 on page 58
`jmq.autocreate.topic`	boolean	`false`	Table 2-9 on page 58
`jmq.cluster.url`	string	`null`	Table 2-10 on page 63
`jmq.keystore.file.dirpath`	string	`JMQ_VARHOME/security`	Table 2-9 on page 58
`jmq.keystore.file.name`	string	`jmqkeystore`	Table 2-9 on page 58
`jmq.keystore.passfile.dirpath`	string	`null`	Table 2-6 on page 51
`jmq.keystore.passfile.enabled`	boolean	`false`	Table 2-6 on page 51
`jmq.keystore.passfile.name`	string	`keypassfile`	Table 2-6 on page 51
`jmq.keystore.password`	string	`null`	Table 2-6 on page 51
`jmq.log.console.output`	string	`ERROR|WARNING`	Table 2-8 on page 54
`jmq.log.console.stream`	string	`ERR`	Table 2-8 on page 54
`jmq.log.file.dirpath`	string	`JMQ_VARHOME/store/`*`brokerName`*`/log`	Table 2-8 on page 54
`jmq.log.file.name`	string	`log.txt`	Table 2-8 on page 54
`jmq.log.file.output`	string	`ALL`	Table 2-8 on page 54
`jmq.log.file.rolloverbytes`	integer (bytes)	`0`	Table 2-8 on page 54
`jmq.log.file.rolloversecs`	integer (seconds)	`604800`	Table 2-8 on page 54
`jmq.log.level`	string	`INFO`	Table 2-8 on page 54

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

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

Property Name	Type	Default Value	Reference
`jmq.message.expiration.interval`	integer (seconds)	`60`	Table 2-4 on page 45
`jmq.message.max_size`	byte string * 0 (no limit)	`70m`	Table 2-4 on page 45
`jmq.metrics.enabled`	boolean	`true`	Table 2-8 on page 54
`jmq.metrics.interval`	integer (seconds)	`0`	Table 2-8 on page 54
`jmq.persist.file.message.fdpool.limit`	integer	`25` (Solaris & Linux) `1024` (Windows)	Table 2-5 on page 48
`jmq.persist.file.message.filepool.cleanratio`	integer	`60`	Table 2-5 on page 48
`jmq.persist.file.message.filepool.limit`	integer	`10000`	Table 2-5 on page 48
`jmq.persist.file.sync.enabled`	boolean	`false`	Table 2-5 on page 48
`jmq.persist.store`	string	`file`	Table 2-5 on page 48
`jmq.portmapper.port`	integer	`7676`	Table 2-3 on page 42
`jmq.queue.deliverypolicy`	string	`single`	Table 2-9 on page 58
`jmq.redelivered.optimization`	boolean	`true`	Table 2-4 on page 45
`jmq.service.activelist`	list	`jms, httpjms, admin`	Table 2-3 on page 42

`jmq.`*`service_name`*`.max_threads`	integer	`1000 for jms`
		`500 for ssljms`
		`500 for httpjms`
		`10 for admin`
			Table 2-3 on page 42

`jmq.`*`service_name`*`.min_threads`	integer	`10 for jms`
		`10 for ssljms`
		`10 for httpjms`
		`4 for admin`
			Table 2-3 on page 42

`jmq.`*`service_name`*`.accesscontrol.enabled`	boolean	`inherits value from system-wide property`	Table 2-6 on page 51

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

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

Property Name	Type	Default Value	Reference
`jmq.`*service_name*`.accesscontrol.file.filename`	string	`inherits value from system-wide property`	Table 2-6 on page 51
`jmq.`*service_name*`.`*protocol name*`.port`	integer	`0`	Table 2-3 on page 42
`jmq.`*service_name*`.authentication.type`	string	`inherits value from system-wide property`	Table 2-6 on page 51
`jmq.swap.percent`	integer (percent)	`50`	Table 2-4 on page 45
`jmq.swap.threshold_count`	integer, 0 (no limit)	`0`	Table 2-4 on page 45
`jmq.swap.threshold_size`	byte string* 0 (no limit)	`70m`	Table 2-4 on page 45
`jmq.system.max_count`	integer, 0 (no limit)	`0`	Table 2-4 on page 45
`jmq.system.max_size`	byte string*, 0 (no limit)	`0`	Table 2-4 on page 45

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

Starting a Broker

To start a broker and configure one or more properties, use the `jmqbroker` command, specifying a valid option. Command-line options override values in the broker configuration files, but only for the current broker session. Command line options are not written to the configuration property file.

The syntax of the `jmqbroker` command is as follows:

```
jmqbroker [[ -Dproperty=value]...]
     [ -backup filename]
     [ -cluster "[broker] [[,broker]...]"
     [ -dbuser username] -dbpassword password]
     [ -javahome path ] | -jrehome path]
     [ -license name]
     [ -loglevel level]
     [ -metrics number]
     [ -name brokername ] [ -port number]
     [ -password keypass] [ -passfile filename]
     [ -reset module]
     [ -restore filename]
     [ -silent] [ -tty]
     [ -version ] [ -vmargs arg [[arg]...]
```

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

```
jmqbroker
```

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

Table 5-2 describes the options to the `jmqbroker` command and describes the configuration properties affected by each option.

Table 5-2 `jmqbroker` Options

Option	Properties Affected	Description
`-D`*property*`=`*value*	Sets system properties. Overrides corresponding property value in configuration file.	Sets the specified property to the specified value. **Caution:** Be careful to check the spelling and formatting of properties set with the D option. If you pass incorrect values, the system will not warn you, and iMQ will not be able to set them.
`-backup` *filename*	None affected.	Backs up a Master Broker's configuration change record to the specified file. Only applicable to broker clusters. See "Backing up the Master Broker's Configuration Change Record" on page 117.
`-cluster "[`*broker*`] [[,`*broker*`]...]"` *broker* is either • *host*[:*port*] • [*host*]:*port*	Sets `jmq.cluster.brokerlist` to the list of brokers to which to connect.	Connects to all the brokers on the specified hosts and ports. This list is merged with the list in the `jmq.cluster.brokerlist` property. If you don't specify a value for *host*, `localhost` is used. If you don't specify a value for *port*, the value 7676 is used. See "Working With Broker Clusters" on page 113 for more information on how to use this option to connect multiple brokers.
`-dbpassword` *password*	`jmq.persist.jdbc.password`	Specifies the password for a plugged-in JDBC-compliant database. See Appendix A, "Setting Up Plugged-in Persistence."
`-dbuser` *userName*	`jmq.persist.jdbc.user`	Specifies the user name for a plugged-in JDBC-compliant database. See Appendix A, "Setting Up Plugged-in Persistence."
`-javahome` *path*	None affected.	Specifies the path to an alternate Java 2 compatible JDK. The default is to use the bundled runtime.
`-jrehome` *path*	None affected.	Specifies the path to a Java 2 JRE.

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

Option	Properties Affected	Description
`-license` [*name*]	None affected.	Specifies the license to load, if different from the default for your iMQ product edition. If you don't specify a license name, this lists all licenses installed on the system. Depending on the installed iMQ edition, the values for *name* are `try` (trial edition), `dev` (developer edition), and `unl` (enterprise edition). See "iMQ Product Editions" on page 22.
`-loglevel` *level*	Sets `jmq.broker.log.level` to the specified level.	Specifies the logging level as being one of `NONE`, `ERROR`, `WARNING`, `INFO`. The default value is `INFO`. For more information, see "Logger" on page 53.
`-metrics` *int*	Sets `jmq.metrics.report.interval` to the specified number of seconds.	Specifies that metrics be reported at an interval specified in seconds.
`-name` *brokerName*	Sets `jmq.instancename` to the specified name.	Specifies the instance name of this broker and uses the corresponding instance configuration file. If you do not specify a broker name, the name of the file is set to `jmqbroker`. **Note**: If you run more than one instance of a broker on the same host, each must have a unique name.
`-passfile` *filename*	Sets `jmq.keystore.passfile.enabled` to `true`. Sets `jmq.keystore.passfile.dirpath` to the path that contains the file. Sets `jmq.keystore.passfile.name` to the name of the file.	Specifies the name of the file from which to read the password for the ssl certificate keystore. For more information, see "" on page 48.
`-password` *pwd*	Sets `jmq.keystore.password` to the specified password.	Specifies the password for the ssl certificate keystore. For more information, see "" on page 48.
`-port` *number*	Sets `jmq.portmapper.port` to the specified number.	Specifies the broker's Port Mapper port number. By default, this is set to 7676. To run two instances of a broker on the same server, each broker's Port Mapper must have a different port number.

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

Option	Properties Affected	Description
`-reset store|props`	None affected.	Reset the broker's persistent store or reset the broker's properties, depending on the argument given. Resetting the broker's persistent store clears out all persistent data, including persistent messages, durable subscriptions, and transaction information. This allows you to start the broker with a clean slate. If you do not want the persistent store to be reset on subsequent starts, you should shut down the broker and then re-start it without using the reset option. For more information, see "Persistence Manager" on page 46. Resetting the broker's properties, deletes the existing instance configuration file (`config.properties`) and replaces it with an empty file.
`-restore` *filename*	None affected.	Replaces the Master Broker's configuration change record with the specified file. See "Restoring the Master Broker's Configuration Change Record" on page 118.
`-silent`	Sets `jmq.log.console.output` to `NONE`.	Turns off logging to the console.
`-tty`	Sets `jmq.log.console.output to ALL`	Specifies that all messages be displayed to the console. By default only `WARNING` and `ERROR` level messages are displayed.
`-version`	None affected.	Displays the version number of the installed product.
`-vmargs arg [[arg]...]`	None affected	Specifies arguments to pass to the Java VM. Separate arguments with spaces. If you want to pass more than one argument or if an argument contains a space, use enclosing quotation marks. For example: `jmqbroker -tty -vmargs "-Xmx128m -Xincgc"`

Working With Broker Clusters

This section describes the properties you use to configure broker clusters, describes a couple of methods of connecting brokers, and explains how you manage clusters.

For an introduction to clusters, see "Multi-Broker Configurations (Clusters)" on page 58.

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

Cluster Configuration Properties

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

Table 5-3 Cluster Properties Summary

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

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

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

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

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

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

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

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

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

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

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

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

Connecting Brokers

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

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

Method 1: No Cluster Configuration File

➤ **To connect brokers into a cluster**

1. Use the `-cluster` option to the `jmqbroker` command that starts a broker, and specify the complete list of brokers (to connect to) as an argument to the `-cluster` option.
2. Do this for each broker you want to connect to the cluster when you start that broker.

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

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

Method 2: Using a Cluster Configuration File

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

Adding Brokers to Clusters

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

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

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

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

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

   ```
   jmqcmd reload cls
   ```

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

Restarting a Broker in a Cluster

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

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

Removing a Broker from a Cluster

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

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

  ```
  jmqbroker -cluster A,B,C
  ```

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

  ```
  jmqbroker -cluster B,C
  ```

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

- If the list of brokers was specified using a cluster configuration file, then you will need to do the following:

  - Remove mention of the broker from the configuration file.
  - Change or remove the `jmq.cluster.url` property for the broker that is being removed so that it no longer uses the common properties.
  - Use the `jmqcmd reload cls` command to force all the brokers to reload their cluster configuration and thereby reconfigure the cluster.

Backing up the Master Broker's Configuration Change Record

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

```
jmqbroker -backup mybackuplog
```

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

Restoring the Master Broker's Configuration Change Record

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

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

   ```
   jmqbroker  -restore mybackuplog
   ```

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

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

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

Logging

This section describes the default logging configuration for the broker and explains how you can change that configuration in order to redirect log information to alternate output channels, to change rollover criteria, and to report broker metrics. For an introduction to logging, see "Logger" on page 53.

Default Logging Configuration

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

```
JMQ_VARHOME/stores/BrokerName/log/
```

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

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

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

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

The broker supports three log categories: `ERROR`, `WARNING`, `INFO` (see Table 2-7 on page 53). Setting a logging level gathers messages for all levels up to and including that level. The default log level is `INFO`. This means that `ERROR`, `WARNING`, and `INFO` messages are logged.

Log Message Format

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

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

Changing the Logging Configuration

All logging-related properties are described in Table 2-8 on page 54.

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

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

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

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

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

Table 5-4 Logger-related Options and Corresponding Properties

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

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

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

Changing the Output Channel

By default, error and warning messages are displayed on the terminal as well as being logged to a log file.

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 `jmqbroker` command.
- To prevent log output from being displayed on the screen, use the `-silent` option to the `jmqbroker` command.
- Use the `jmq.log.file.output` property to specify which categories of logging information should be written to the log file. For example,

  ```
  jmq.log.file.output=ERROR
  ```

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

  ```
  jmq.log.console.output=INFO
  ```

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

Changing Rollover Criteria

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

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

  ```
  jmq.log.file.rolloversecs=864000
  ```

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

  ```
  jmq.log.file.rolloverbytes=500000
  ```

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

Logging Broker Performance Metrics

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

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

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

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

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

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

The following shows sample metrics information:

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

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

Table 5-5 Metrics Gathered for Connection Services

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

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

Table 5-6 Metrics Gathered for Each Broker

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

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

Chapter 6

Broker and Application Management

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

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

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

- iMQ messages are routed to their receivers or subscribers by way of broker destinations. You are responsible for creating these destinations on the broker.
- iMQ allocates and maintains resources for durable subscribers even when clients that have durable subscriptions become inactive. You use the iMQ Command tool to get information about durable subscriptions and to destroy durable subscriptions in order to save iMQ resources.

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

iMQ Command Utility

The iMQ Command utility allows you to manage the broker and the services it provides. This section describes the basic format of `jmqcmd` commands and provides a summary of `jmqcmd` options. Subsequent sections explain how you use these commands to accomplish specific tasks.

Format of Commands

The general format of `jmqcmd` commands is as follows:

```
jmqcmd [subcommand argument ] options
```

or

```
jmqcmd subcommand argument [options ]
```

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

```
jmqcmd restart bkr -v
```

Summary of jmqcmd Options

Table 6-1 lists the options to the `jmqcmd` command. For a discussion of their use, see the following task-based sections.

Table 6-1 `jmqcmd` Options

Option	Description
`-b` *hostName:port*	Specifies the name of the broker's host and the number of its port. The default value is `localhost:7676`. To specify port only: `-b :7878` To specify name only: `-b somehost`
`-c` *"clientID"*	Specifies the ID of the durable subscriber to a topic. For more information, see "Managing Durable Subscriptions" on page 142.
`-d` *topicName*	Specifies the name of the topic. Used with the `list dur` and `destroy dur` subcommands. See "Managing Durable Subscriptions" on page 142.

Table 6-1 `jmqcmd` Options *(Continued)*

Option	Description
`-int interval`	Specifies the interval, in seconds, at which `jmqcmd` displays broker metrics. (Used with the `metrics` subcommand.)
`-m` *metricType*	Specifies the type of metric information to display. Type can be one of the following `ttl` Total of messages in and out of the broker (default). `rts` Provides the same information as ttl, but specifies the number of messages per second. `cxn` Connections, virtual memory heap, threads Use this option with the `metrics bkr` or `metrics svc` subcommand. The following command displays cxn-type metrics for the default broker every five seconds. `jmqcmd metrics bkr -m cxn -int 5`
`-n` *targetName*	Specifies the name of the target. Depending on the subcommand, this might be the name of a physical destination, a service, or a durable subscription. For information about its use with destination management, see "Managing Destinations" on page 137.
`-o` *attribute=value*	Specifies the value of an attribute. Depending on the subcommand argument, this might be the attribute of a broker, service, or destination. For information about its use with destination management, see "Managing Destinations" on page 137.
`-p` *password*	Specifies your (the administrator's) password. If you omit this value, you will be prompted for it.
`-t` *destinationType*	Specifies the type of a destination: `t` (topic) or `q` (queue).
`-u` *name*	Specifies your (the administrator's) name. If you omit this value, you will be prompted for it.

You *must* specify the options for host name and port number (`-b`), user name (`-u`) and password (`-p`) *each time* you issue a `jmqcmd` subcommand. If you don't specify user name and password information, you will be prompted for them. You don't have to specify the host name and port number if you want to use the default values.

Prerequisites to Using jmqcmd

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

- Start the broker using the `jmqbroker` command.

  See "Starting a Broker" on page 109. You can use the iMQ Command utility only to administer brokers that are already running; you cannot use it to start a broker.

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

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

  When you install iMQ, a default flat-file user repository is copied to the installation directory. The file is called `JMQ_VARHOME/security/passwd`. The repository is shipped with two entries: one for an admin user and one for a guest user. These entries allow you to connect to the broker without doing any additional work. For example, if you are just testing iMQ, you can run the utility using your default user name and password (`admin/admin`).

  If you are setting up a production system, you will need to do some additional work to authenticate and authorize users. You also have the option of using an existing LDAP directory server for your user repository. For more information, see "Authenticating Users" on page 162.

Entering any `jmqcmd` command automatically connects you to the broker specified on the command line (or to the default broker); there is no separate `jmqcmd` command for connecting to the broker.

Examples

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

```
jmqcmd query bkr -u admin -p admin
```

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

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

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

Controlling the Broker's State

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

- Pausing the broker

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

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

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

- Resuming the broker

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

  ```
  jmqcmd resume bkr
  ```

- Shutting down the broker

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

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

- Restarting the broker

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

  ```
  jmqcmd restart bkr
  ```

Table 6-2 summarizes the `jmqcmd` subcommands used to control the broker. Remember that you must specify the broker host name and port number unless you are targeting the broker running on `localhost` at port 7676.

Table 6-2 `jmqcmd` Subcommands Used to Control the Broker

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

Querying and Updating Broker Properties

The iMQ Command utility includes subcommands that you can use to get information about the broker and to update broker properties Table 6-3 lists these subcommands. Note that updates to the broker are automatically written to the broker's instance configuration file.

Table 6-3 `jmqcmd` Subcommands Used to Get Information and to Update Broker

Subcommand Sytnax	Description
`query bkr -b` *hostName:port*	Lists the current settings of updatable properties of the default broker or a broker at the specified host and port. Also shows the list of running brokers that are connected to the specified broker.
`reload cls`	Forces all the brokers in a cluster to reload the `jmq.cluster.brokerlist` property and update cluster information. See "Adding Brokers to Clusters" on page 116 for more information.
`update bkr -b` *hostName:port* `-o` *attr=val*	Changes the specified attributes for the default broker or a broker at the specified host and port.
`metrics bkr -b` *hostName:port* `[-m metricType] [-int interval]`	Displays broker metrics for the default broker or a broker at the specified host and port. Use the -m option to specify the type of metric to display: `ttl` Total of messages in and out of the broker (default). `rts` Provides the same information as ttl, but specifies the number of messages per second. `cxn` Connections, virtual memory heap, threads Use the -int option to specify the interval (in seconds) at which to display the metrics. The default is 5 seconds

Remember that you must specify the broker host name and port number when using any of the subcommands listed in Table 6-3 unless you are targeting the broker running on `localhost` at port 7676

You can use the `update` subcommand to update any of the broker properties listed in Table 6-4.

Table 6-4 Broker Properties

Properties	Description
`jmq.autocreate.topic`	Specifies whether a broker is allowed to auto-create a topic destination. `True` by default.
`jmq.autocreate.queue`	Specifies whether a broker is allowed to auto-create a queue destination. `True` by default.
`jmq.portmapper.port`	Specifies the number of the port mapper port. Default is `7676`.
`jmq.log.level`	Specifies the log level as one of the following: `NONE`, `ERROR`, `WARNING`, `INFO`. Default is `INFO`.
`jmq.log.file.rolloverbytes`	Specifies the maximum size of the log file before it is rolled over. A value of 0 means no rollover based on file size. The default value is 0.
`jmq.log.file.rolloversecs`	The age (in seconds) before the log file is rolled over. A value of 0 means no rollover based on the age of the file. The default value is 604800 (7 days).
`jmq.swap.threshold_count`	Specifies the maximum number of messages in memory before swapping to disk takes place. A value of 0 means no limit. By default, there is no limit.
`jmq.swap.threshold_size`	Specifies the maximum number of bytes in memory before swapping to disk takes place. A value of 0 means no limit. Default is `70m`.
`jmq.system.max_count`	Specifies the maximum number of messages in memory and disk. A value of 0 means no limit. By default, there is no limit.
`jmq.system.max_size`	Specifies the maximum total size of messages in memory and disk. A value of 0 means no limit. By default, there is no limit.
`jmq.message.max_size`	Specifies the maximum size of a message in bytes. Default is `70m`.
`jmq.cluster.url`	Specifies the location of the cluster configuration file. For more information, see"Cluster Configuration Properties" on page 113.

Managing Connection Services

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

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

For an overview of iMQ connection services, see "Connection Services" on page 40.

Table 6-5 lists the `jmqcmd` subcommands that control connection services. If no host name or port is specified, they are assumed to be `localhost`, `7676`.

Table 6-5 `jmqcmd` Subcommands Used to Manage Connection Services

Subcommand Syntax	Description
`list svc -b` *hostName:port*	Lists all connection services on the default broker or on a broker at the specified host and port.
`metrics svc -n` *serviceName* `-b` *hostName:port* `[-m` *metricType*`] [-int` *interval*`]`	List metrics for the specified service on the default broker or on a broker at the specified host and port. Use the -m option to specify the type of metric to display: `ttl` Total of messages in and out of the broker (default). `rts` Provides the same information as ttl, but specifies the number of messages per second. `cxn` Connections, virtual memory heap, threads Use the -int option to specify the interval (in seconds) at which to display the metrics. The default is 5 seconds.
`query svc -n` *serviceName* `-b` *hostName:port*	Displays information about the specified service running on the default broker or on a broker at the specified host and port.
`pause svc -n` *serviceName* `-b` *hostName:port*	Pauses the specified service running on the default broker or on a broker at the specified host and port. You cannot pause the admin service.

Table 6-5 `jmqcmd` Subcommands Used to Manage Connection Services *(Continued)*

Subcommand Syntax	Description
`resume svc -n` *serviceName* `-b` *hostName:port*	Resumes the specified service running on the default broker or on a broker at the specified host and port.
`update svc -n` *serviceName* `-b` *hostName:port* `-o` *attr=val*	Updates the specified attribute of the specified service running on the default broker or on a broker at the specified host and port. For a description of service attributes, see Table 6-7 on page 136.

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

Table 6-6 Connection Services Supported by an iMQ Broker

Service Name	Service Type	Connection Type (Transport Protocol)
jms	NORMAL (JMS message delivery)	tcp
ssljms	NORMAL (JMS message delivery)	ssl (over tcp)
httpjms	NORMAL (JMS message delivery)	http
admin	ADMIN	tcp

Listing Services

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

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

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

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

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

```
jmqcmd list svc
```

The command will output information like the following:

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

Host                  Primary Port
localhost             7676

Service Name      Port Number       Service State
jms               33983 (dynamic)   RUNNING
admin             33984 (dynamic)   RUNNING
ssljms            dynamic           UNKNOWN
httpjms           0 (dynamic)       RUNNING

Successfully listed services.
```

Querying and Updating Service Properties

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

```
jmqcmd query svc -n jms
```

This produces output like the following:

```
Querying the service where:

Service Name
jms

On the broker specified by:

Host                          Primary Port
localhost                     7676


Service Name                              jms
Port Number                               42019 (dynamic)
Service State                             RUNNING
Min Number of Threads                     50
Max Number of Threads                     150
Current Number of Allocated Threads       120
Current Number of Connections             20

Successfully queried the service.
```

You can use the `update` subcommand to change the value of one or more of the service attributes listed in Table 6-7.

Table 6-7 Service Attributes

Attribute	Description
`port`	The port assigned to the service to be updated.
`minThreads`	The minimum number of threads assigned to the service.
`maxThreads`	The maximum number of threads assigned to the service.

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

```
jmqcmd update svc -n jms -o "minThreads=20"
```

Pausing and Resuming a Service

To pause any service other than the admin service, use a command like the following:

```
jmqcmd pause svc -n serviceName
```

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

```
jmqcmd resume svc -n serviceName
```

Managing Destinations

All iMQ messages are routed to their consumer clients by way of destinations, queues and topics, created on a particular broker. You are responsible for managing these destinations on the broker. This involves using the iMQ Command utility to create and destroy destinations, to list destinations, to display information about destinations, and to purge messages. For an introduction to destinations, see "Physical Destinations" on page 55.

Table 6-8 provides a summary of the `jmqcmd` destination subcommands. Remember to specify the host name and port of the broker where these services are running if this is not the default (`localhost:7676`) broker.

Table 6-8 `jmqcmd` Subcommands Used to Manage Destinations

Subcommand	Description
`list dst`	Lists all destinations, except for temporary destinations (see "Temporary Destinations" on page 58).
`create dst -t` *type* `-n` *name* `[-o` *att=val*`]` `[-o` *att=val1*`]...]`	Creates a destination of the specified type, with the specified name, and the specified attributes. Destination names must contain only alphanumeric characters (no spaces) and can begin with an alphabetic character or the character "_"
`destroy dst -t` *type* `-n` *name*	Destroys the destination of the specified type and name.
`purge dst -t` *type* `-n` *name*	Purge messages at the destination with the specified type and name.
`query dst -t` *type* `-n` *name*	Lists information about the destination of the specified type and name.
`update dst -t` *type* `-n` *name* `-o` *att=val* `[[-o` *att=val1*`]...]`	Updates the value of the specified attributes at the specified destination. The attribute name may be one of the following: `maxTotalMsgBytes`, `maxBytesPerMsg`, or `maxNumMsgs`.

Creating Destinations

When creating a destination, you must specify its type (topic or queue) and, if needed, specify values for the destination's attributes. Default values for these attributes are set in the broker's configuration file (see "Configuration Files" on page 103.)

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

Table 6-9 describes the attributes that can be set for each type of destination when you create the destination.

Table 6-9 Destination Attributes

Destination Type	Attribute	Default Value	Description
Queue	`queueDeliveryPolicy`	`Single`	Describes the algorithm used to route messages. Values are f = Failover r = Round robin s = Single
Queue	`maxTotalMsgBytes`	`0` (unlimited)	Maximum total size in bytes of messages allowed in the queue.
Queue	`maxNumMsgs`	`0` (unlimited)	Maximum number of messages allowed in the queue
Queue	`maxBytesPerMsg`	`0` (unlimited)	Maximum size of any single message allowed in the queue.
Topic	`maxBytesPerMsg`	`0` (unlimited)	Maximum size of any single message posted to the topic.

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

  ```
  jmqcmd create dst -n myQueue -t q -o "queueDeliveryPolicy=f"
  ```

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

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

  ```
  jmqcmd create dst -n myTopic -t t -o "maxByetsPerMsg=5000"
  ```

Getting Information About Destinations

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

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

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

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

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

The `list` command also lists temporary destinations. These are destinations created by client applications that need a destination at which to receive replies to messages sent to other clients. You cannot destroy these destinations; they can only be destroyed by API calls made by the client application when there are no more active message consumers.

Updating Destinations

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

```
jmqcmd update dst -t q -n myQueue -o "maxBytesPerMsg=1000"
             -o maxNumMsgs=2000
```

See Table 6-9 on page 139 for a list of the attributes that you can update.

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

Purging Destinations

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

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

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

```
jmqcmd purge dst -n myTopic -t t
```

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

```
jmqbroker -reset store
```

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

Destroying Destinations

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

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

Managing Durable Subscriptions

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

You can use two subcommands to manage durable subscriptions: `list`, which displays all the durable subscriptions to a particular topic, and `destroy`, which allows you to destroy a durable subscription.

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

```
jmqcmd list dur -d SPQuotes
```

For each durable subscription to a topic, the `list` subcommand returns the name of the durable subscription, the client ID of the user, the number of messages queued to this topic, and the state of the durable subscription (active/inactive). For example:

```
Listing all the durable subscriptions on the topic myTopic
on the broker specified by:
-------------------------
Host         Primary Port
-------------------------
localhost    7676

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

Successfully listed durable subscriptions.
```

You can use the information returned from the `list` command to identify a durable subscription you might want to destroy. Use the name of the subscription and the client ID to identify the subscription. For example:

```
jmqcmd destroy dur -n myDurable -c "myClientID"
```

Chapter 7

Managing Administered Objects

The use of administered objects enables the development of client applications that are portable to other JMS providers. *Administered objects* are objects that encapsulate provider-specific configuration and naming information. These objects are normally created by an iMQ administrator and used by client applications to obtain connections to the broker, which are then used to send messages to and receive messages from physical destinations.

iMQ provides two administration tools for creating and managing administered objects: the command line iMQ Object Manager utility (`jmqobjmgr`) and the GUI iMQ Administration Console. These tools enable you to do the following:

- Add or delete administered objects to an object store.
- List existing administered objects.
- Query and display information about an administered object.
- Modify an existing administered object in the object store.

This chapter explains how you use the iMQ Object Manager utility (`jmqobjmgr`) to perform these tasks. For information about the Administration Console, see Chapter 4, "iMQ Administration Console Tutorial."

About Object Stores

Administered objects are placed in a readily available object store where they can be accessed by client applications through a JNDI lookup. There are two types of object stores you can use with iMQ: a standard LDAP directory server or a file-system object store.

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

File-system Store iMQ also supports a file-system object store implementation. While the file-system object store is not fully tested and is therefore not recommended for production systems, it has the advantage of being very easy to use in development environments. Rather than setting up an LDAP server, all you have to do is create a directory on your local file system. Any user with access to that directory can use iMQ administration tools to create and manage administered objects.

Administered Objects

Administered objects may be of two kinds: connection factories and destinations. *Connection factories* are objects used by client applications to create a connection to a broker. *Destinations* are objects that client applications use to identify the destination to which a producer is sending messages or from which a consumer is retrieving messages.

Depending on the domain (point-to-point or publish/subscribe) in which they are used, connection factories and destinations have a specific type. In the point-to-point domain, a connection factory is called a queue connection factory and a destination is called a queue. In the publish and subscribe domain, a connection factory is called a topic connection factory and a destination is called a topic (see Table 1-1 on page 31).

For an overview of administered objects, see"iMQ Administered Objects" on page 66.

iMQ Object Manager Utility (jmqobjmgr)

The iMQ Object Manager utility (`jmqobjmgr`) includes the following subcommands:

Table 7-1 `jmqobjmgr` Subcommands

Subcommand	Description
`add`	Adds an administered object to the object store.
`delete`	Deletes an administered object from the object store.
`list`	Lists administered objects in the object store.
`query`	Displays information about the specified administered object.
`update`	Modifies an existing administered object in the object store.

The iMQ Object Manager utility (`jmqobjmgr`) includes the following options:

Table 7-2 `jmqobjmgr` Options

Option	Description
`-i` *fileName*	Specifies the name of a java property file containing the command to execute and other information required by the command (lookup name, object store attributes, object type, object attributes).
`-l` *name*	Specifies the JNDI lookup name of the administered object. This name must be unique in the object store's context.
`-o` *attribute=value*	Specifies the attributes of the administered object.
`-j` *attribute=value*	Specifies the attributes of the JNDI object store.
`-r` *read-only*	Specifies whether the administered object should be created as a read-only object. A value of `true` creates the administered object as a read-only object. Client applications cannot modify the attributes of administered objects that are read-only. Set to `false` by default.

Table 7-2 `jmqobjmgr` Options *(Continued)*

Option	Description
`-t` *type*	Specifies the type of the iMQ administered object: q = queue t = topic qf = queueConnectionFactory tf = topicConnectionFactory

The following section describes information that you need to provide when working with any `jmqobjmgr` subcommand.

Required Information

When performing most tasks related to administered objects, the administrator must specify the following information as options to `jmqobjmgr` subcommands:

- The type of the administered object:

  `topic, queue, topicConnectionFactory, or queueConnectionFactory.`

- The JNDI lookup name of the administered object:

  This is the logical name that will be used in the client code to refer to the administered object (using JNDI) in the object store.

- The attributes of the administered object:

  For queues and topics, the name of the physical destination on the broker: this is the name that was specified with the `-n` option to the `jmqcmd create` subcommand. If you do not specify the name, the default name of `Untitled_Destination_Object` will be used.

  For connection factories, this includes among others, the host name and port number of the broker to which the client will connect. If you do not specify this information, the local host and default port number (7676) are used. The section "Object Attributes" on page 147 explains how you specify object attributes.

- The attributes of the JNDI object store:

  This information depends on whether you are using a file-system store or LDAP server, but must include the following attributes:

  - The type of JNDI implementation (initial context attribute). For example, file-system or LDAP.
  - The location of the administered object in the object store (provider URL attribute), that is, its "folder" as it were.
  - The user name, password, and authorization type, if any, required to access the object store.

  For more information about object store attributes see "Object Store Attributes" on page 149.

Object Attributes

The attributes of an administered object are specified using attribute-value pairs. The following sections describe these attributes.

Connection Factory Objects

Connection factory administered objects have the attributes listed in Table 7-3. The two attributes you are primarily concerned with are `JMQBrokerHostPort` and `JMQBrokerHostName`, which you use to specify the broker to which the client application will establish a connection. The section, "Adding a Connection Factory" on page 154, explains how you specify these attributes when you add a connection factory administered object to your object store.

For more detailed information on connection factory attributes, see the *iMQ Developer's Guide* and the JavaDoc API documentation for the iMQ class `com.sun.messaging.ConnectionConfiguration`.

Table 7-3 Connection Factory Attributes

Attribute/property name	Type	Default Value
`JMQAckOnAcknowledge`	String	`not specified`
`JMQAckOnProduce`	String	`not specified`
`JMQAckTimeout`	String	`0 milliseconds`
`JMQBrokerHostName`	String	`localhost`
`JMQBrokerHostPort`	String	`7676`

Table 7-3 Connection Factory Attributes *(Continued)*

Attribute/property name	Type	Default Value
JMQConfiguredClientID	String	not specified
JMQConnectionType	String	TCP
JMQConnectionURL	String	http://localhost/servlet /HttpTunnelServlet
JMQDefaultPassword	String	guest
JMQDefaultUsername	String	guest
JMQDisableSetClientID	String	false
JMQFlowControlCount	String	100
JMQFlowControlIsLimited	String	false
JMQFlowControlLimit	String	1000
JMQLoadMaxToServerSession	String	false
JMQQueueBrowserMax MessagesPerRetrieve	String	1000
JMQQueueBrowserRetrieve Timeout	String	60,000 milliseconds
JMQReconnect	String	false
JMQReconnectDelay	String	30,000 milliseconds
JMQReconnectRetries	String	0
JMQSetJMSXAppID	String	false
JMQSetJMSXConsumerTXID	String	false
JMQSetJMSXProducerTXID	String	false
JMQSetJMSXRcvTimestamp	String	false
JMQSetJMSXUserID	String	false
JMQSSLIsHostTrusted	String	true

Destination Objects

The destination administered object that identifies a physical topic or queue destination has the two attributes listed in Table 7-4. The attribute you are primarily concerned with is JMQDestinationName. The section, "Adding a Topic or Queue" on page 155, explains how you specify these attributes when you add a destination administered object to your object store.

For more information, see "iMQ Administered Objects" on page 66 and the JavaDoc API documentation for the iMQ class `com.sun.messaging.DestinationConfiguration`.

Table 7-4 Destination Attributes

Attribute/property name	Type	Default
`JMQDestinationName`	String*	`Untitled_Destination_Object`
`JMQDestinationDescription`	String	`A Description for the Destination Object`

*. Destination names must contain only alphanumeric characters (no spaces) and can begin with an alphabetic character or the characters "_" and "$".

Object Store Attributes

The attributes of the object store are specified using the `-j` option and consist of attribute-value pairs. In general, you must specify the following attributes:

Initial Context and Location Information

The format for these entries differs depending on whether you are using a file-system store or LDAP server.

File-system store As an example of using a file-system store, create a folder called *MyObjstore* on the C drive, and specify the following values for the initial context and location attributes, respectively:

```
-j "java.naming.factory.initial=
          com.sun.jndi.fscontext.RefFSContextFactory"

-j "java.naming.provider.url=file://C:/MyObjStore"
```

LDAP server As an example of using an LDAP server, specify the following values for the initial context and location attributes, respectively:

```
-j "java.naming.factory.initial=
          com.sun.jndi.ldap.LdapCtxFactory"

-j "java.naming.provider.url=ldap://mydomain.com:389/o=JMQ"
```

Security Information (LDAP Only)

The format for these entries differs depending on the LDAP provider. You should also consult the documentation provided with your LDAP implementation to determine whether security information is required on all operations or only on operations that change the stored data.

Security attributes look like this:

```
-j "java.naming.security.principal=
      uid=fooUser, ou=People, o=JMQ"
-j "java.naming.security.credentials=fooPasswd"
-j "java.naming.security.authentication=simple"
```

Table 7-5 describes these entries:

Table 7-5 Security Attributes for the Object Store

Attribute	Description
...principal	The identity of the principal for authenticating the caller to the service.The format of this entry depends on the authentication scheme. If this property is unspecified, the behavior is determined by the service provider.
...credentials	The credentials of the principal for authenticating the caller to the service. The value of the property depends on the authentication scheme. For example, it could be a hashed password, clear-text password, key, certificate, and so on. If this property is unspecified, the behavior is determined by the service provider.
...authentication	Security level to use. Its value is one of the following key words: `none`, `simple`, `strong`. If this property is unspecified, the behavior is determined by the service provider. If you specify `simple`, jmqobjmgr will prompt for any missing principal or credential values. This will allow you a more secure way of providing identifying information.

Using Input Files

The `jmqobjmgr` command allows you to specify the name of an input file that uses java property file syntax to represent all or part of the `jmqobjmgr` subcommand clause.

Using an input file with the iMQ Object Manager utility (`jmqobjmgr`) is especially useful to specify object store attributes, which are likely to be the same across multiple invocations of `jmqobjmgr` and which normally require a lot of typing. Using an input file can also allow you to avoid a situation in which you might otherwise exceed the maximum number of characters allowed for the command line.

The general syntax for a `jmqobjmgr` input file is as follows:

```
cmdtype=[ add | delete | list | query | update ]
obj.type=[ q | t | qf | tf ]
obj.lookupName=lookup name
obj.attrs.objAttrName1=value1
obj.attrs.objAttrName2=value2
obj.attrs.objAttrNameN=valueN
...
objstore.attrs.objStoreAttrName1=value1
objstore.attrs.objStoreAttrName2=value2
objstore.attrs.objStoreAttrNameN=valueN
...
```

As an example of how you can use an input file, consider the following `jmqobjmgr` command:

```
jmqobjmgr add
       -t qf
       -l "cn=myQCF"
       -o "JMQBrokerHostName=foo"
       -o "JMQBrokerHostPort=777"
       -j "java.naming.factory.initial=
                com.sun.jndi.ldap.LdapCtxFactory"
       -j "java.naming.provider.url=
                ldap://mydomain.com:389/o=JMQ"
       -j "java.naming.security.principal=
                uid=fooUser, ou=People, o=JMQ"
       -j "java.naming.security.credentials=fooPasswd"
       -j "java.naming.security.authentication=simple"
```

This command can be encapsulated in a file, say `MyCmdFile`, that has the following contents:

```
    cmdtype=add
    obj.type=qf
    obj.lookupName=cn=myQCF
    obj.attrs.JMQBrokerHostName=foo
    obj.attrs.JMQBrokerHostPort=777
    objstore.attrs.java.naming.factory.initial=\
              com.sun.jndi.ldap.LdapCtxFactory
    objstore.attrs.java.naming.provider.url=\
              ldap://mydomain.com:389/o=JMQ
    objstore.attrs.java.naming.security.principal=\
              uid=fooUser, ou=People, o=JMQ
    objstore.attrs.java.naming.security.credentials=fooPasswd
    objstore.attrs.java.naming.security.authentication=simple
```

You can then use the -i option to pass this file to the iMQ Object Manager utility (`jmqobjmgr`):

```
jmqobjmgr -i MyCmdFile
```

You can also use the input file to specify some options, while using the command line to specify others. This allows you to use the input file to specify parts of the subcommand clause that is the same across many invocations of the utility. For example, the following command specifies all the options needed to add a connection factory administered object, except for those that specify where the administered object is to be stored.

```
jmqobjmgr add
        -t qf
        -l "cn=myQCF"
        -o "JMQBrokerHostName=foo"
        -o "JMQBrokerHostPort=777"
        -i MyCmdFile
```

In this case, the file `MyCmdFile` would contain the following definitions:

```
    objstore.attrs.java.naming.factory.initial=\
              com.sun.jndi.ldap.LdapCtxFactory
    objstore.attrs.java.naming.provider.url=\
              ldap://mydomain.com:389/o=JMQ
    objstore.attrs.java.naming.security.principal=\
              uid=fooUser, ou=People, o=JMQ
    objstore.attrs.java.naming.security.credentials=fooPasswd
    objstore.attrs.java.naming.security.authentication=simple
```

Additional examples of input files can be found at the following location:

```
JMQ_HOME/examples/jmqobjmgr
```

Adding and Deleting Administered Objects

This section explains how you add administered objects for connection factories and topic or queue destinations to the object store.

Adding a Connection Factory

To enable client applications to obtain a connection to the broker, you add an administered object that represents the type of connections the client applications want: a topic connection factory or a queue connection factory

To add a queue connection factory, use a command like the following:

```
jmqobjmgr add
  -t qf
  -l "cn=myQCF"
  -o "JMQBrokerHostName=myHost"
  -o "JMQBrokerHostPort=7272"
  -j "java.naming.factoryinitial=
              com.sun.jndi.ldap.LdapCtxFactory"
  -j "java.naming.provider.url=ldap://mydomain.com:389/o=JMQ"
  -j "java.naming.security.principal=
        uid=fooUser, ou=People, o=JMQ"
  -j "java.naming.security.credentials=fooPasswd"
  -j "java.naming.security.authentication=simple"
```

The preceding command creates an administered object whose lookup name is `cn=myQCF` and which connects to a broker running on `myHost` and listens on port 7272. The administered object is stored in an LDAP server.

NOTE If you are using an LDAP server to store the administered object, it is important that you assign a lookup name that has the prefix "cn=" as in the example above. You specify the lookup name with the `-l` option. You do not have to use this prefix if you are using a file-system object store.

You can accomplish the same thing by specifying an input file as an argument to the `jmqobjmgr` command. For more information, see "Using Input Files" on page 151.

Adding a Topic or Queue

To enable client applications to access physical destinations on the broker, you add administered objects that identify these destinations, to the object store.

It is best to first create the physical destinations before adding the corresponding administered objects to the object store. Use the iMQ Command utility (`jmqcmd`) to create the physical destinations on the broker that are identified by destination administered objects in the object store. For information about creating physical destinations, see "Managing Destinations" on page 137.

The following command adds an administered object that identifies a topic destination whose lookup name is `myTopic` and whose physical destination name is `TestTopic`. The administered object is stored in an LDAP server.

```
jmqobjmgr add
     -t t
     -l "cn=myTopic"
     -o "JMQDestinationName=TestTopic"
     -j "java.naming.factory.initial=
            com.sun.jndi.ldap.LdapCtxFactory"
     -j "java.naming.provider.url=
            ldap://mydomain.com:389/o=JMQ"
     -j "java.naming.security.principal=
            uid=fooUser, ou=People, o=JMQ"
     -j "java.naming.security.credentials=fooPasswd"
     -j "java.naming.security.authentication=simple"
```

This is the same command, only the administered object is stored in a Solaris file system:

```
jmqobjmgr add
      -t t
      -l "cn=myTopic"
      -o "JMQDestinationName=TestTopic"
      -j "java.naming.factory.initial=
             com.sun.jndi.fscontext.RefFSContextFactory"
      -j "java.naming.provider.url=
             file:/home/foo/jmq_admin_objects"
```

In the LDAP server case, as an example, you could use an input file, `MyCmdFile`, to specify the subcommand clause. The file would contain the following text:

```
cmdtype=add
    obj.type=t
    obj.lookupName=cn=myTopic
    obj.attrs.JMQDestinationName=TestTopic
    objstore.attrs.java.naming.factory.initial=
            com.sun.jndi.fscontext.RefFSContextFactory
    objstore.attrs.java.naming.provider.url=
            file:/home/foo/jmq_admin_objects
    objstore.attrs.java.naming.security.principal=
            uid=fooUser, ou=People, o=JMQ
    objstore.attrs.java.naming.security.credentials=fooPasswd
    objstore.attrs.java.naming.security.authentication=simple
```

Use the -i option to pass the file to the `jmqobjmgr` command:

```
jmqobjmgr -i MyCmdFile
```

NOTE If you are using an LDAP server to store the administered object, it is important that you assign a lookup name that has the prefix "cn=" as in the example above. You specify the lookup name with the `-l` option. You do not have to use this prefix if you are using a file-system object store.

Adding a queue object is exactly the same, except that you specify `q` for the `-t` option.

Deleting Administered Objects

Use the `delete` subcommand to delete an administered object. You must specify the lookup name of the object, its type, and its location.

The following command deletes an administered object for a topic whose lookup name is `cn=myTopic` and which is stored on an LDAP server.

```
jmqobjmgr delete
     -t t
     -l "cn=myTopic"
     -j "java.naming.factory.initial=
            com.sun.jndi.ldap.LdapCtxFactory"
     -j "java.naming.provider.url=
            ldap://mydomain.com:389/o=JMQ"
     -j "java.naming.security.principal=
            uid=fooUser, ou=People, o=JMQ"
     -j "java.naming.security.credentials=fooPasswd"
     -j "java.naming.security.authentication=simple"
```

Getting Information

Use the `list` and `query` subcommands to list administered objects in the object store and to display information about an individual object.

Listing Administered Objects

Use the `list` subcommand to get a list of all administered objects or to get a list of all administered objects of a specific type. The following sample code assumes that the administered objects are stored in an LDAP server.

The following command lists all objects.

```
jmqobjmgr list
    -j "java.naming.factory.initial=
            com.sun.jndi.ldap.LdapCtxFactory"
    -j "java.naming.provider.url=
            ldap://mydomain.com:389/o=JMQ"
    -j "java.naming.security.principal=
            uid=fooUser, ou=People, o=JMQ"
    -j "java.naming.security.credentials=fooPasswd"
    -j "java.naming.security.authentication=simple"
```

The following command lists all objects of type `queue`.

```
jmqobjmgr list
  -t q
  -j "java.naming.factory.initial=
           com.sun.jndi.ldap.LdapCtxFactory"
  -j "java.naming.provider.url=
           ldap://mydomain.com:389/o=JMQ"
  -j "java.naming.security.principal=
           uid=fooUser, ou=People, o=JMQ"
  -j "java.naming.security.credentials=fooPasswd"
  -j "java.naming.security.authentication=simple"
```

Information About a Single Object

Use the `query` subcommand to get information about an administered object. You must specify the object's lookup name and the attributes of the object store containing the administered object (such as initial contact and location).

In the following example, the `query` subcommand is used to display information about an object whose lookup name is `myTopic`.

```
jmqobjmgr query
  -l "cn=myTopic"
  -j "java.naming.factory.initial=
         com.sun.jndi.ldap.LdapCtxFactory"
  -j "java.naming.provider.url=
         ldap://mydomain.com:389/o=JMQ"
  -j "java.naming.security.principal=
         uid=fooUser, ou=People, o=JMQ"
  -j "java.naming.security.credentials=fooPasswd"
  -j "java.naming.security.authentication=simple"
```

Updating Administered Objects

You use the `update` command to modify the attributes of administered objects. You must specify the lookup name and location of the object. You use the -o option to modify attribute values.

This command changes the attributes of an administered object that represents a topic connection factory:

```
jmqobjmgr update
 -t tf
 -l "cn=MyTCF"
 -o JMQReconnectRetries=3
 -j "java.naming.factory.initial=
        com.sun.jndi.ldap.LdapCtxFactory"
 -j "java.naming.provider.url=
        ldap://mydomain.com:389/o=JMQ"
 -j "java.naming.security.principal=
        uid=fooUser, ou=People, o=JMQ"
 -j "java.naming.security.credentials=fooPasswd"
 -j "java.naming.security.authentication=simple"
```

Chapter 8

Security Management

This chapter explains how to perform tasks related to security, these include authentication, authorization, and encryption.

Authenticating Users You are responsible for maintaining a list of users, their groups, and passwords in a user repository. The first part of this chapter explains how you create, populate, and manage that repository. For an introduction to iMQ security, see "" on page 48.

Authorizing Users You are responsible for editing a properties file that maps the user's access to broker operations to the user's name or group. The second part of this chapter explains how you can customize this properties file.

Encryption: Setting Up SSL Services Using a connection service based on the Secure Socket Layer (SSL) standard allows you to encrypt messages sent between clients and broker. For an introduction to how iMQ handles encryption, see "Encryption" on page 50. The last part of this chapter explains how to set up an SSL-based connection service and provides additional information about using SSL.

Authenticating Users

When a user attempts to connect to the broker, the broker authenticates the user by inspecting the name and password provided, and grants the connection if they match those in a user repository that the broker is configured to consult. This repository might be one of the following:

- a flat-file repository that is shipped with iMQ

  To enable user authentication, you must populate the broker's user repository each user's, name, password, and the name of the user's group. If you are using the iMQ flat-file user repository, you can use the iMQ User Manager (`jmqusermgr`) utility to populate and manage the user repository.

  For more information on setting up and managing the user repository, see "Using a Flat-File User Repository" on page 162.

- an LDAP server

  This could be an existing or new LDAP directory server that uses the LDAP v2 or v3 protocol for your user repository. If you are using an LDAP user repository, you will need to use the tools provided by the LDAP vendor to populate and manage the user repository. For more information, see "Using an LDAP Server for a User Repository" on page 168.

Using a Flat-File User Repository

iMQ provides a flat-file user repository and a command line tool, iMQ User Manager (`jmqusermgr`) that you can use to populate and manage the flat-file user repository. The following sections describe the flat-file user repository, its initial entries, and how you populate and manage that repository.

When iMQ is installed, a default flat-file repository is copied to the installation directory. The file is called `JMQ_VARHOME/security/passwd`.

The repository is shipped with two entries (rows) already defined, as illustrated in the table below.

Table 8-1 Initial Entries in User Repository

User Name	Password	Group	State
`admin`	`admin`	`admin`	`active`
`guest`	`guest`	`anonymous`	`active`

These initial entries allow the iMQ broker to be used immediately after installation without any intervention by the administrator. In other words, no initial user/password setup is required for the iMQ broker to be used.

The initial `guest` user entry allows JMS clients to connect to the broker using the default `guest` user name and password (for testing purposes, for example).

The initial `admin` user entry allows you to use `jmqcmd` commands to administer the broker using the default `admin` user name and password. It is recommended that you update this initial entry to change the password.

- On Solaris, after installation, the user repository file can only be written to by users with superuser privileges, consistent with the operating system policies for controlling access to the file using the permissions attributes of the file.
- On Windows (NT and 2000), after installation, the user repository file can be written to by any user because the operating system does not control access to files using user name-based permission attributes.

After installing the broker, you can use the iMQ User Manager utility to populate the user repository. The broker does not need to be configured or started before this is done. The only requirement for using the iMQ User Manager utility is that it be run on the host where the broker is installed. The following sections explain how you populate and manage the repository used by the broker.

iMQ User Manager Subcommands and Options

Table 8-2 lists the `jmqusermgr` subcommands.

Table 8-2 `jmqusermgr` Subcommands

Subcommand	Description
`add -u` *name* `-p` *passwd* `[-g` *group*`] [-s]`	Add a user and associated password to the repository, and optionally specify the user's group
`delete -u` *name* `[-s] [-f]`	Delete a user from the repository.
`list [-u` *name*`]`	List information about one or more users.
`update -u` *name* `-p` *passwd* `[-a` *state*`] [-s] [-f]` `update -u` *name* `-a` *state* `[-p` *passwd*`] [-s] [-f]`	Update a user's password and/or state.

Table 8-3 lists the options to the `jmqusermgr` command.

Table 8-3 `jmqusermgr` Options

Option	Description
`-a true | false`	Specify whether the user's state should be active. A value of true means that the state is active. This is the default.
`-f`	Perform action without user confirmation
`-h`	Display help.
`-p` *passwd*	Specify the user's password.
`-g admin | user | anonymous`	Specify the user group.
`-s`	Set silent mode.
`-u` *name*	Specify the user name.
`-v`	Display version info

Groups

When adding a user entry to the repository, the administrator has the option of specifying one of three predefined groups for the user: `admin`, `user`, or `anonymous`. If no group is specified, the default group `user` is assigned.

- The *admin* group is for broker administrators. Users who are assigned this group can, by default, configure, administer, and manage the broker. The administrator can assign more than one user to the admin group.
- The *user* group is for normal (non-administrative) JMS client applications. Most iMQ client applications will access the broker authenticated in the user group. As such, client applications, can produce messages to and consume messages from all topics and queues, or can browse messages in any queue by default.
- The *anonymous* group is for JMS client applications who do not wish to use a user name that is known to the broker (possibly because the application does not know of a real user name to use). This is analogous to the anonymous account present in most FTP servers. The administrator can assign only one user to the anonymous group at any one time. It is expected that you will restrict the access privileges of this group as compared to the user group through access control or that you will remove the user from this group at deployment time.

In order to change a user's group, the administrator must delete the user entry and then add another entry for the user, specifying the new group.

You can specify access rules that define what operations the members of that group may perform. For more information, see "Authorizing Users: the Access Control Properties File" on page 170.

States

When the administrator adds a user to the repository, the user's state is active by default. To make the user inactive, the administrator must use the update command. For example, the following command makes the user JoeD inactive:

```
jmqusermgr update -u JoeD -a false
```

Entries for users that have been rendered inactive are retained in the repository; however, inactive users cannot open new connections. If a user is inactive and the administrator adds another user who has the same name, the operation will fail. The administrator must delete the inactive user entry or change the new user's name or use a different name for the new user. This prevents the administrator from adding duplicate names or passwords.

Format of User Names and Passwords

User names and passwords must follow these guidelines:

- The user name and passwords may not contain the characters listed in Table 8-4.

Table 8-4 Invalid Characters for User Names and Passwords

Character	Description
*	Asterisk
,	Comma
:	Colon

- The user name and passwords may not contain a new line or carriage return as characters.
- If the name or password contains a space, the entire name or password must be enclosed in quotation marks.

- The name or password must be at least one character long.
- There is no limit on the length of passwords or user names—except for that imposed by the command shell on the maximum number of characters that can be entered on a command line.

Populating and Managing the User Repository

Use the `add` subcommand to add a user to the repository. For example, the following command adds the user, `Katharine` with the password `sesame`.

```
jmqusermgr add -u Katharine -p sesame -g user
```

Use the `delete` subcommand to delete a user from the repository. For example, the following command deletes the user, `Bob`:

```
jmqusermgr delete -u Bob
```

Use the `update` subcommand to change a user's password or state. For example, the following command changes Katharine's password to `alladin`:

```
jmqusermgr update -u Katharine -p alladin
```

To list information about one or more users, use the `list` command. The following command shows information about the user named `isa`:

```
jmqusermgr list -u isa
```

```
---------------------------------
User Name    Group     Active State
---------------------------------
isa          admin    true
```

The following command lists information about all users:

```
jmqusermgr list
```

```
-------------------------------------
User Name    Group        Active State
-------------------------------------
testuser3    user         true
testuser2    user         true
testuser1    user         true
isa          admin        true
admin        admin        true
guest        anonymous    true
testuser5    user         false
testuser4    user         false
```

Changing the Default Administrator Password

For the sake of security, you must change the default password of `admin` to one that is only known to you. You need to use the `jmqusermgr` tool to do this.

The following command changes the default password to `grandpoobah`.

```
jmqusermgr update -u admin -p grandpoobah
```

You can quickly confirm that this change is in effect, by running any of the command line tools when the broker is running. For example, the following command should work,

```
jmqcmd list svc -u admin -p grandpoobah
```

While using the old password should fail.

After changing the password, you should supply the new password when using any of the administration tools, including the administration console.

Using an LDAP Server for a User Repository

If you want to use an LDAP server for your user repository, you must set certain broker properties in the instance configuration file. These properties enable the broker to query the LDAP server for information about users and groups when a user attempts to connect to the broker or perform certain operations. The instance configuration file is located at

```
JMQ_VARHOME/stores/brokerName/props/config.properties
```

➤ **To edit the configuration file to use an LDAP server**

1. Specify that you are using an LDAP user repository by setting the following property:

   ```
   jmq.authentication.basic.user_repository=ldap
   ```

2. Set the `jmq.authentication.type` property to determine whether a password should be passed from client to broker in base64 encoding (`basic`) or in MD5 digest (`digest`). When using an LDAP directory server for a user repository, you must set the authentication type to `basic`. For example,

   ```
   jmq.authentication.type=basic
   ```

3. You must also set the broker properties that control LDAP access. These are described in Table 8-5. iMQ uses JNDI APIs to communicate with the LDAP directory server. Consult JNDI documentation for more information on syntax and on terms referenced in these properties. iMQ 2.0 uses a Sun JNDI LDAP provider and uses simple authentication.

Table 8-5 LDAP-related Properties

Property	Description
`jmq.user_repository.ldap.server`	The host:port for the LDAP server. Host specifies the fully qualified DNS name of the host running the directory server. Port specifies the port number that the directory server is using for communications.
`jmq.user_repository.ldap.bindDN`	The distinguished name that the broker will use to bind to the directory server for a search. If the directory server allows anonymous searches, this property does not need to be assigned a value.
`jmq.user_repository.ldap.bindPW`	The password associated with the distinguished name for the broker. If the directory server allows anonymous searches, this property does not need to be assigned a value.

Table 8-5 LDAP-related Properties *(Continued)*

Property	Description
`jmq.user_repository.ldap.base`	The directory base for user entries.
`jmq.user_repository.ldap.uidattr`	The provider-specific attribute identifier whose value uniquely identifies a user. For example: `uid`, `cn`, etc.
`jmq.user_repository.ldap.usrfilter`	A JNDI search filter (a search query expressed as a logical expression). By specifying a search filter for users, the broker can narrow the scope of a search and thus make it more efficient. For more information, see the JNDI tutorial at the following location: http://java.sun.com/products/jndi/tutorial. This property does not have to be set.
`jmq.user_repository.ldap.grpsearch`	A boolean specifying whether you want to enable group searches. Consult the documentation provided by your LDAP provider to determine whether you can associate users into groups. Note that nested groups are not supported in iMQ 2.0. Default: `false`
`jmq.user_repository.ldap.grpbase`	The directory base for group entries.
`jmq.user_repository.ldap.gidattr`	The provider-specific attribute identifier whose value is a group name.
`jmq.user_repository.ldap.memattr`	The attribute identifier in a group entry whose values are the distinguished names of the group's members.
`jmq.user_repository.ldap.grpfiltler`	A JNDI search filter (a search query expressed as a logical expression). By specifying a search filter for groups, the broker can narrow the scope of a search and thus make it more efficient. For more information, see the JNDI tutorial at the following location. http://java.sun.com/products/jndi/tutorial This property does not have to be set.
`jmq.user_repository.ldap.timeout`	An integer specifying (in seconds) the time limit for a search. By default this is set to 180 seconds.
`jmq.user_repository.ldap.ssl.enabled`	A boolean specifying whether the broker should use the SSL protocol when talking to an LDAP server. This is set to `false` by default.

See the broker's default.properties file for a sample (default) LDAP user-repository-related properties setup.

4. If necessary, you need to edit the users/groups and rules in the access control properties file. For more information about the use of access control property files, see "Authorizing Users: the Access Control Properties File" on page 170.

5. If you want the broker to communicate with the LDAP directory server over SSL during connection authentication and group searches, you need to activate SSL in the LDAP server and then set the following properties in the broker configuration file:

   - Specify a secure port for the LDAP user repository property. For example:

     ```
     jmq.user_repository.ldap.server=myhost:7878
     ```

   - Set the broker `property jmq.user_repository.ldap.ssl.enabled` to `true`.

Authorizing Users: the Access Control Properties File

After connecting to the broker, the user may want to produce a message, consume a message at a destination, or browse messages at a queue destination. When the user attempts to do this, the broker checks an *access control properties file* (ACL file) to see whether the user is authorized to perform the operation. The ACL file contains rules that specify which operations a particular user (or group of users) is authorized to perform. By default, all authenticated users are allowed to produce and consume messages at any destination. You can edit the access control properties file to restrict these operations to certain users and groups.

The ACL file is used whether user information is placed in a flat-file repository or in an LDAP repository. A default ACL properties file is installed along with the broker. Its name is `accesscontrol.properties` and it is placed by the installer in the following directory: `JMQ_VARHOME/security`.

The ACL file is formatted like a Java properties file. It starts by defining the version of the file and then specifies access control rules in three sections:

- connection access control
- destination access control
- destination auto-create access control

The `version` property defines the version of the ACL properties file; you may not change this entry.

```
version=JMQFileAccessControlModel/100
```

The three sections of the ACL file that specify access control are described below, following a description of the basic syntax of access rules and an explanation of how permissions are calculated.

Access Rules Syntax

In the ACL properties file, access control defines what access specific users or groups have to protected resources like destinations and connection services. Access control is expressed by a rule or set of rules, with each rule presented as a Java property:

The basic syntax of these rules is as follows:

resourceType.resourceVariant.operation.access.principalType = *principals*

Table 8-6 describes the elements of syntax rules.

Table 8-6 Syntactic Elements of Access Rules

Element	Description
resourceType	One of the following: `connection`, `queue` or `topic`.
resourceVariant	An instance of the type specified by *resourceType*. For example, myQueue. The wild card character (*)may be used to mean all connection service types or all destinations.
operation	Value depends on the kind of access rule being formulated.
access	One of the following: `allow` or `deny`.
principalType	One of the following: `user` or `group`. For more information, see "Groups" on page 164.
principals	Who may have the access specified on the left-hand side of the rule. This may be an individual user or a list of users (comma delimited) if the `principalType` is `user`; it may be a single group or a list of groups (comma delimited list) if the `principalType` is `group`. The wild card character (*)may be used to represent all users or all groups.

Here are some examples of access rules:

- The following rule means that all users may send a message to the queue named q1.

  ```
  queue.q1.produce.allow.user=*
  ```

- The following rule means that any user may send messages to any queue.

  ```
  queue.*.produce.allow.user=*
  ```

NOTE To specify non-ASCII user, group, or destination names, you must use Unicode escape (`\uXXXX`) notation. If you have edited and saved the ACL file with these names in a non-ASCII encoding, you can convert the file to ASCII with the Java `native2ascii` tool. For more detailed information, see `http://java.sun.com/j2se/1.3/docs/guide/intl/faq.html`

Permission Computation

The following principles are applied when computing the permissions implied by a series of rules:

- Specific access rules override general access rules. After applying the following two rules, all can send to all queues, but Bob cannot send to tq1.

  ```
  queue.*.produce.allow.user=*
  queue.tq1.produce.deny.user=Bob
  ```

- Access given to an explicit *principal* overrides access given to a * *principal*. The following rules deny Bob the right to produce messages to tq1, but allow everyone else to do it.

  ```
  queue.tq1.produce.allow.user=*
  queue.tq1.produce.deny.user=Bob
  ```

- The * *principal* rule for users overrides the corresponding * *principal* for groups. For example, the following two rules allow all authenticated users to send messages to tq1.

  ```
  queue.tq1.produce.allow.user=*
  queue.tq1.produce.deny.group=*
  ```

- Access granted a user overrides access granted to the user's group. In the following example, if Bob is a member of User, he will be denied permission to produce messages to tq1, but all other members of User will be able to do so.

```
queue.tq1.produce.allow.group=User
queue.tq1.produce.deny.user=Bob
```

- Any access permission not explicitly granted through an access rule is implicitly denied. For example, if the ACL file contained no access rules, all users would be denied all operations.
- Deny and allow permissions for the same user or group cancel themselves out. For example, the following two rules result in Bob not being able to browse t1:

```
queue.q1.browse.allow.user=Bob
queue.q1.browse.deny.user=Bob
```

  The following two rules result in the group User not being able to consume messages at q5.

```
queue.q5.consume.allow.group=User
queue.q5.consume.deny.group=User
```

- When multiple same left-hand rules exist, only the last entry takes effect.

Connection Access Control

The connection access control section in the ACL properties file contains access control rules for the broker's connection services. The syntax of connection access control rules is as follows:

```
connection.resourceVariant.access.principalType = principals
```

Two values are defined for *resourceVariant*: `NORMAL` and `ADMIN`. By default all users can have access to the `NORMAL` type, but only those users whose group is `admin` may have access to `ADMIN` type connection services.

You can edit the connection access control rules to restrict a user's connection access privileges. For example, the following rules deny Bob access to `NORMAL` but allow everyone else:

```
connection.NORMAL.deny.user=Bob
```

```
connection.NORMAL.allow.user=*
```

You can use the asterisk (*) character to specify all authenticated users or groups.

You may not create your own service type; you must restrict yourself to the predefined types specified by the constants `NORMAL` and `ADMIN`.

Destination Access Control

The destination access control section of the access control properties file contains destination-based access control rules. These rules determine who (users/groups) may do what (operations) where (destinations). The types of access that are regulated by these rules include sending messages to a queue, publishing messages to a topic, receiving messages from a queue, subscribing to a topic, and browsing a messages in a queue.

By default, any user or group can have all types of access to any destination. You can add more specific destination access rules or edit the default rules. The rest of this section explains the syntax of destination access rules, which you must understand to write your own rules.

The syntax of destination rules is as follows:

resourceType.resourceVariant.operation.access.principalType = principals

Table 8-7 describes these elements:

Table 8-7 Elements of Destination Access Control Rules

Component	Description
resourceType	Must be one of `queue` or `topic`.
resourceVariant	A destination name or all destinations (*), meaning all queues or all topics.
operation	Must be one of `produce`, `consume`, or `browse`.
access	Must be one of `allow` or `deny`.
principalType	Must be one of `user` or `group`.

Access can be given to one or more users and/or one or more groups.

The following examples illustrate different kinds of destination access control rules:

- Allow all users to send messages to any queue destinations.

  ```
  queue.*.produce.allow.user=*
  ```

- Deny any member of the group `user` to subscribe to the topic Admissions.

  ```
  topic.Admissions.consume.deny.group=user
  ```

Destination Auto-Create Access Control

The final section of the ACL properties file, includes access rules that specify for which users and groups the broker will auto-create a destination.

When a user creates a producer or consumer at a destination that does not already exist, the broker will create the destination if the broker's auto-create property has been enabled and if the physical destination does not already exist.

By default, any user or group has the privilege of having a destination auto-created by the broker. This privilege is specified by the following rules:

```
queue.create.allow.user=*

topic.create.allow.user=*
```

You can edit the ACL file to restrict this type of access.

The general syntax for destination auto-create access rules is as follows:

resourceType.`create`.*access.principalType* `=` *principals*

Where *resourceType* is either `queue` or `topic`.

For example, the following rules allow the broker to auto-create topic destinations for everyone except Snoopy.

```
topic.create.allow.user=*
topic.create.deny.user=Snoopy
```

Note that the effect of destination auto-create rules must be congruent with that of destination access rules. For example, if you 1) change the destination access rule to forbid any user from sending a message to a destination but 2) enable the auto-creation of the destination, the broker *will* create the destination if it does not exist but it will *not* deliver a message to it.

Encryption: Working With an SSL Service

Using a connection service based on the Secure Socket Layer (SSL) standard allows you to encrypt messages sent between clients and broker.

Setting Up an SSL Service

This section explains how you set up an SSL-based connection service. The procedure includes the following steps:

- Generate a self-signed certificate.
- Enable the SSL Service in the broker.
- Start the Broker.
- Configure and Run the Client.

1. Generate a Self-Signed Certificate

SSL Support in iMQ 2.0 is oriented toward securing on-the-wire data with the assumption that the client is communicating with a known and trusted server. Therefore the preferred method of deploying SSL in iMQ 2.0 is to use self-signed certificates.

Run the `jmqkeytool` utility to generate a self-signed certificate. Enter the following at the command prompt:

```
jmqkeytool
```

The utility will prompt you for the information it needs. (On Unix systems you may need to run `jmqkeytool` as the superuser (root) in order to have permission to create the keystore.)

First, `jmqkeytool` prompts you for a password, then it prompts you for some organizational information, and then it prompts you for confirmation. After it receives the confirmation, it pauses while it generates a key pair. It then asks you for a password to lock the particular key pair; you must enter Return in response to this prompt: this makes the key password the same as the keystore password.

NOTE Remember the password you provide, you will need to provide this password later to the broker (when you start it) so it can open the keystore.

Running `jmqkeytool` runs the JDK `keytool` utility to generate a self-signed certificate and to place it in iMQ's keystore, located at

```
JMQ_VARHOME/security/jmqkeystore
```

The keystore is in the same keystore format as that supported by the JDK1.2 `keytool`.

2. Enable the SSL Service in the Broker

To enable the SSL service in the broker, you need to add `ssljms` to the `jmq.service.activelist` property.

1. Open the broker's configuration file. You can find it at the following location:

   `JMQ_VARHOME/stores/`*brokerInstance*`/props/config.properties`

   brokerInstance is the name of the broker.

2. Add the `ssljms` value to the `jmq.service.activelist` property:

   ```
   jmq.service.activelist=jms,admin,httpjms,ssljms
   ```

For a complete listing of ssl-related broker properties, see Table 2-6 on page 51.

3. Start the Broker

Start the broker and provide the keystore password. You can provide the password in any one of the following ways:

- Use the `-password` option to the `jmqbroker` command:

  ```
  jmqbroker -password mypassword
  ```

- Put the password in a file and pass the location of the file to the `jmqbroker` command:

  ```
  echo mypasssword > /tmp/passwordfile
  jmqbroker -passfile /tmp/passwordfile
  ```

- Allow the broker to prompt you for the password when it starts up

  ```
  jmqbroker
  Please enter Keystore password: mypassword
  ```

- Set the password in the broker's configuration file using the following property:

  ```
  jmq.keystore.password=mypassword
  ```

  The configuration file is located at

  `JMQ_VARHOME/stores/`*brokerInstance*`/props/config.properties`

  brokerInstance is the name of the broker.

4. Configure and Run the Client

Finally, you need to make sure the client has the necessary jar files in its classpath and you need to pass certain information to it when you start it.

1. Make sure the client has the following jar files in its class path:

   ```
   jmq.jar, jms.jar, jndi.jar, jsse.jar, jnet.jar, jcert.jar
   ```

2. Start the client and connect to the broker's SSL service by entering a command like the following:

   ```
   java -DJMQConnectionType=SSL
   ```

   Setting `JMQConnectionType` tells the connection to use SSL.

Regenerating a Key Pair

You may need to regenerate a key pair in order to solve certain problems; for example:

- You forgot the keystore password.
- The SSL service fails to initialize when you start the broker and you get the exception:
  `java.security.UnrecoverableKeyException: Cannot recover key`.

  This exception may result from the fact that you provided a key password that was different from the keystore password.

➤ **To regenerate a key pair**

1. Remove the broker's keystore

   ```
   rm JMQ_VARHOME/security/jmqkeystore
   ```

2. Rerun `jmqkeytool` to generate a key pair as described in "1. Generate a Self-Signed Certificate" on page 176.

Resource Consumption

When you start a broker or client with SSL, you might notice that it consumes a lot of cpu cycles for a few seconds. This is because iMQ uses JSSE (Java Secure Socket Extension) to implement SSL. JSSE uses `java.security.SecureRandom()` to generate random numbers. This method takes a significant amount of time to create the initial random number seed, and that is why you are seeing increased cpu usage. After the seed is created, the cpu level will drop to normal.

Appendix A

Setting Up Plugged-in Persistence

This appendix explains how to set up a broker to use plugged-in persistence to access a JDBC-accessible data store.

Introduction

iMQ brokers include a Persistence Manager component that manages the writing and retrieval of persistent information (see "Persistence Manager" on page 46). The Persistence Manager is configured by default to access a built-in, file-based data store, but you can reconfigure it to plug in any data store accessible through a JDBC-compliant driver.

To configure a broker to use plugged-in persistence, you need to set a number of JDBC-related properties in the broker instance configuration file. You also need to create the appropriate database schema for performing iMQ persistence operations. iMQ provides a utility, iMQ Database Manager (`jmqdbmgr`), which uses your JDBC driver and broker configuration properties to create and manage the plugged-in database.

The procedure described in this appendix is illustrated using, as an example, the Cloudscape DBMS bundled with the Java 2 SDK Enterprise Edition (J2EE SDK is available for download from java.sun.com). The example uses Cloudscape's embedded version (instead of the client/server version). In the procedures, instructions are illustrated using path names and property names from the Cloudscape example. They are identified by the word "Example:"

Other examples can be found at the following location:

```
JMQ_HOME/examples/jdbc
```

Plugging In a JDBC-accessible Data Store

It takes just a few steps to plug in a JDBC-accessible data store.

➤ **To plug in a JDBC-accessible data store**

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

   See the properties documented in Table A-1 on page 181.

2. Place a copy or a symbolic link to your JDBC driver jar file in the following path:

   ```
   JMQ_VARHOME/lib/
   ```

   Example (copy):

   ```
   % cp j2sdk_install_directory/lib/cloudscape/cloudscape.jar
   JMQ_VARHOME/lib
   ```

   Example (symbolic link):

   ```
   % ln -s j2sdk_install_directory/lib/cloudscape/cloudscape.jar
   JMQ_VARHOME/lib
   ```

3. Create the database schema needed for iMQ persistence.

   Use the `jmqdbmgr create all` command (for an embedded database) or the `jmqdbmgr create tbl` command (for an external database). See "The iMQ Database Manager Utility (jmqdbmgr)" on page 183.

   Example:

   ```
   % cd JMQ_HOME/bin
   % ./jmqdbmgr create all
   ```

JDBC-related Broker Configuration Properties

The broker's instance configuration file is located in

```
JMQ_VARHOME/stores/brokerName/props/config.properties
```

If the file does not yet exist, you have to start up the broker using the `-name brokerName` option, for iMQ to create the file.

Table A-1 presents the configuration properties that you need to set when plugging in a JDBC- accessible data store. You set these properties in the instance configuration file (`config.properties`) of each broker instance using plugged-in persistence. The table includes values you would specify for the Cloudscape DBMS example.

Table A-1 JDBC-related Properties

Property Name	Description
`jmq.persist.store`	Specifies a file-based or JDBC-based data store. *Example*: `jdbc`
`jmq.persist.jdbc.brokerid (optional)`	Specifies a broker instance identifier that is appended to database table names to make them unique in the case where more than one broker instance is using the same database as a persistent data store. (Usually not needed in the case of an embedded database, which stores data for only one broker instance.) The identifier must be an alphanumeric string whose length does not exceed the maximum table name length, minus 12, allowed by the database. *Example:* not needed for Cloudscape
`jmq.persist.jdbc.driver`	Specifies the java class name of the JDBC driver to connect to the database. *Example*: `COM.cloudscape.core.JDBCDriver`
`jmq.persist.jdbc.opendburl`	Specifies the database URL for opening a connection to an existing database. *Example*: `jdbc:cloudscape:JMQ_VARHOME/stores/brokerName/dbstore/jmqdb`

Table A-1 JDBC-related Properties *(Continued)*

Property Name	Description
`jmq.persist.jdbc.createdburl (optional)`	Specifies the database URL for opening a connection to create a database. (Only specified if the database is to be created using `jmqdbmgr`.) *Example*: `jdbc:cloudscape:JMQ_VARHOME/stores/`*brokerName*`/dbstore/jmqdb;create=true`
`jmq.persist.jdbc.closedburl (optional)`	Specifies the database URL for shutting down the current database connection when the broker is shutdown. *Example* (required for Cloudscape): `jdbc:cloudscape:;shutdown=true`
`jmq.persist.jdbc.user (optional)`	Specifies the user name used to open a database connection, if required. For security reasons, the value can be specified instead using command line options: `jmqbroker -dbuser` and `jmqdbmgr -u`
`jmq.persist.jdbc.password (optional)`	Specifies password used to open a database connection, if required. For security reasons, the value can be specified instead using command line options: `jmqbroker -dbpassword` and `jmqdbmgr -p`

As with all broker configuration properties, values can be set using the `-D` command line option. If a database requires certain database specific properties to be set, these also can be set using the `-D` command line option when starting the broker (`jmqbroker`) or the Database Manager utility (`jmqdbmgr`).

Example:

For the Cloudscape embedded database example, instead of specifying the absolute path of a database in database connection URLs (as those shown in Table A-1 examples), the -D command line option can be used to define the Cloudscape system directory:

```
-Dcloudscape.system.home=JMQ_VARHOME/stores/brokerName/dbstore
```

In that case the URLs to create and open a database can be specified simply as:

```
jmq.persist.jdbc.createdburl=jdbc:cloudscape:jmqdb;create=true
```

and

```
jmq.persist.jdbc.opendburl=jdbc:cloudscape:jmqdb
```

respectively.

The iMQ Database Manager Utility (jmqdbmgr)

iMQ provides a Database Manager utility (`jmqdbmgr`) for setting up the schema needed for iMQ persistence. The utility can also be used to delete the iMQ database tables should the tables become corrupted or should you wish to use a different database as a data store.

If an embedded database is used and it is to be created under the `JMQ_VARHOME/stores/`*brokerName*`/` directory, it is recommended that it should be created under

`JMQ_VARHOME/stores/`*brokerName*`/dbstore/`*dabatabseName*.

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 `jmqdbmgr` command, when used to create an embedded database, should be run by the user who will be running the broker.

Table A-2 lists the `jmqdbmgr` subcommands.

Table A-2 `jmqdbmgr` Subcommands

Subcommand	Description
`create all`	Create a new database and iMQ persistent storage schema. This command is used on an embedded database system, and when used, the property `jmq.persist.jdbc.createdburl` needs to be specified.
`create tbl`	Create the iMQ persistent storage schema in an existing database system. This command is used on an external database system.
`delete tbl`	Deletes the existing iMQ database tables in the current persistent storage database.
`recreate tbl`	Deletes the existing iMQ database tables in the current persistent storage database and then re-creates the iMQ persistent storage schema.

Table A-3 lists the options to the `jmqdbmgr` command.

Table A-3 `jmqdbmgr` Options

Option	Description
`-D`*property*`=`*value*	Sets the specified property to the specified value.
`-b` *name*	Specify the broker instance name and use the corresponding instance configuration file.
`-h`	Display help.
`-p` *passwd*	Specify the database password.
`-u` *name*	Specify the database user name.
`-v`	Display version information

Appendix B

HTTP Support

iMQ includes HTTP support, which allows client applications to communicate with the broker using HTTP protocol instead of direct TCP connections. This appendix describes the architecture used to implement this support and explains the setup work needed to allow clients to use HTTP connections for iMQ messaging.

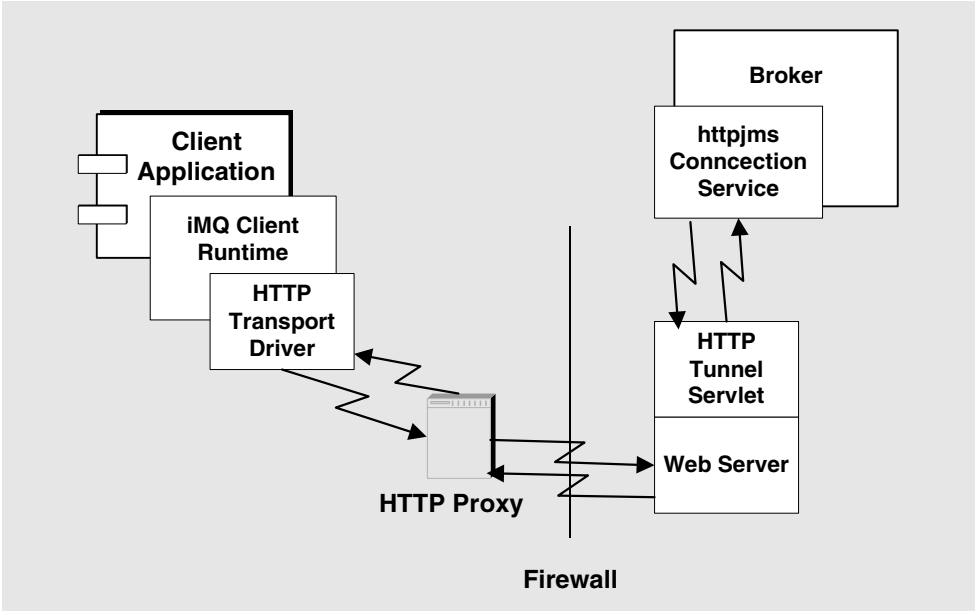
HTTP Support Architecture

iMQ messaging can be run on top of HTTP connections. Because HTTP connections are normally allowed through firewalls, this allows cleint applications to be eparated from a broker by a firewall.

Figure B-1 shows the principal components involved in providing HTTP support.

- On the client side, an HTTP transport driver encapsulates the iMQ message into an HTTP request and makes sure that these requests are sent to the Web server in the correct sequence.
- The client application can use an HTTP proxy server to communicate with the broker if necessary. The proxy's address is specified using command line options when starting the client application. See "Using an HTTP Proxy" on page 188 for more information.
- An HTTP tunnel servlet (bundled with iMQ) is loaded in the web server and used to pull JMS messages out of client HTTP requests before forwarding them to the broker. The HTTP tunnel servlet also sends broker messages back to the client in response to HTTP pull requests made by the client's HTTP transport driver. A single HTTP tunnel servlet can be used to access multiple brokers.
- On the broker side, the httpjms connection service unwraps and demultiplexes incoming messages from the HTTP tunnel servlet.

Figure B-1 HTTP Support Architecture



Implementing HTTP Support

The following sections describe the steps you need to take to implement HTTP support, which consists mainly of the following

- Deploying the HTTP tunnel servlet on a web server
- Configuring the broker's httpjms connection service.
- Configuring the connection factory administered object needed by the client to use the HTTP protocol.

Deploying the HTTP Tunnel Servlet on a Web Server

The `jmqservlet.jar` file contains all the classes needed by the HTTP tunnel servlet and is located in the `JMQ_HOME/lib` directory. Any web server with servlet 2.x support can be used to load this servlet. The servlet class name is `com.sun.messaging.jmq.transport.httptunnel.servlet.HttpTunnelServlet`.

The web server must be able to see the `jmqservlet.jar` file. If you are planning to run the web server and the broker on different hosts, you should place a copy of the `jmqservlet.jar` file in a location where the web server can access it.

You also need to configure the web server to load this servlet on startup (see "Example: Configuring a HTTP Tunnel Servlet" on page 189).

You can verify the servlet setup by accessing the HTTP tunnel servlet URL using a web browser. It should display status information.

It is also recommended that you disable your web server's access logging feature in order to improve performance.

Configuring the httpjms Connection Service

HTTP support is enabled in the default configuration for an iMQ 2.0 broker. After starting, the broker looks for a web server and HTTP tunnel servlet running on its host machine. However, you can configure the following default httpjms connection service properties to access a remote tunnel servlet:

- `jmq.httpjms.http.servletHost=localhost`

  Change this value, if necessary, to specify the name of the host on which the web server and HTTP tunnel servlet are running.

- `jmq.httpjms.http.servletPort=9000`

  Change this value to specify the port number that the broker uses to access the HTTP tunnel servlet.

- `jmq.httpjms.http.pullPeriod=-1`

  This property affects client behavior, even though it is set on the broker. The `pullPeriod` specifies the interval, in seconds, between HTTP requests made by each client to pulling messages from the HTTP tunnel servlet. If the value is zero or negative, the client keeps one HTTP request pending at all times, ready to pull messages from the servlet as fast as possible. With a large number of clients, this can be a heavy drain on web server resources and the server may

become unresponsive. In such cases, you should set the `pullPeriod` property to a positive number of seconds. This sets the time the client's HTTP transport drivers waits before making subsequent pull requests. Setting the value to a positive number conserves web server resources at the expense of the response times observed by clients.

Setting up a Client Connection

You must define the following connection factory attributes to implement HTTP support:

- Set the `JMQConnectionType` attribute to `HTTP`
- Set the `JMQConnectionURL` to the HTTP tunnel servlet URL

You can define these attributes in one of the following ways:

- Using a JMS API call to set the attributes of a connection factory after you create it programmatically.
- Using the `-D` option to the command that launches the client application.
- Using the `-o` option to the `jmqobjmgr` command that creates the connection factory administered object.

Using an HTTP Proxy

If you are using an HTTP proxy to access the HTTP tunnel servlet:

- Set `http.proxyHost` system property to the proxy server host name.
- Set `http.proxyPort` system property to the proxy server port number.

You can set these properties using the `-D` option to the command that launches the client application

Using a Single Servlet to Access Multiple Brokers

You do not need to configure multiple web servers and servlet instances if you are running multiple brokers. You can share a single web server and HTTP tunnel servlet instance among concurrently running brokers. In order to do this, you must configure the `JMQConnectionURL` connection factory attribute as shown below:

`http://`*`HTTP_Tunnel_Servlet_URL`*`?ServerName=`*`hostName`*`:`*`brokerName`*

Where *hostName* is the broker host name and *brokerName* is the name of the specific broker instance you want your client to access.

To check that you have entered the correct strings for *hostName* and *brokerName*, generate a status report for the HTTP tunnel servlet by accessing the servlet URL from a browser. The report lists all brokers being accessed by the servlet.

Example: Configuring a HTTP Tunnel Servlet

This section describes how you configure a HTTP tunnel servlet on the iPlanet Web Server, FastTrack Edition 4.1 using the browser-based administration GUI.

Adding a Servlet

➤ **To add a tunnel servlet**

1. Select the Servlets tab.
2. Select "Configure Servlet Attributes."
3. Specify a name for the tunnel servlet in the Servlet Name field.
4. Set the Servlet Code (class name) field to the following value:

   ```
   com.sun.messaging.jmq.transport.httptunnel.servlet.
   HttpTunnelServlet
   ```

5. Enter the complete path to the `jmqservlet.jar` in the Servlet Classpath field. For example:

   ```
   /opt/SUNWjmq/lib/jmqservlet.jar
   ```

6. By default, the servlet listens for TCP connections from the brokers on port 9000. To use a different port number enter the following in the Servlet Args field.

   `serverPort=`*`number`*

Configuring a Servlet Virtual Path (Servlet URL)

➤ **To configure a virtual path (servlet URL) for a tunnel servlet**

1. Select the Servlets tab.
2. Select "Configure Servlet Virtual Path Translation."

3. Set the Virtual Path field.

   For example, if you want the URL to be `http://`*servername*`:`*port*`/jmq/servlet`, enter the following string in the Virtual Path field.

   ```
   /jmq/servlet
   ```

4. Set the Servlet Name field to the same value as in Step 3 in "Adding a Servlet" on page 189.

Loading a Servlet

➤ **To load the tunnel servlet into the web browser**

1. Select the Servlets tab.
2. Select "Configure Global Attributes."
3. In the Startup Servlets field, enter the same server name value as in Step 3 in "Adding a Servlet" on page 189.

Disabling a Server Access Log

You do not have to disable the server access log, but you will obtain better performance if you do.

➤ **To disable the server access log**

1. Select the Status tab.
2. Select The Log Preferences Page.
3. Use the Log client accesses control to disable logging.

Appendix C

Using a Broker as a Windows Service

This appendix explains how you use the `jmqsvcadmin` utility to install, query, and remove a broker running as a Windows Service (on NT or Windows 2000)

Running a Broker as a Windows Service

You have the option of installing a broker as a Windows service when you install iMQ 2.0. You can also use the `jmqsvcadmin` utility to install a broker as an NT service after you have installed iMQ 2.0.

Installing a broker as an NT service means that it will start at system startup time and run in the background until you shut down. Consequently, you do not use the `jmqbroker` command to start the broker--unless, you want to start an additional instance. To pass any start-up options to the broker, you can use the `-args` argument to the `jmqsvcadmin` command (see Table C-1 on page 192) and specify exactly the same options you would have used for the `jmqbroker` command (see "Starting a Broker" on page 109). Use the `jmqcmd` utility to control broker operations as usual.

When running as an NT service, the Task Manager lists the broker as two executable processes. The first is `jmqbrokersvc.exe`, which is the native NT service wrapper. The second is the Java runtime that is actually running the broker.

Using the jmqsvcadmin Command

You can use the `jmqsvcadmin` command to install, query, and remove the broker (running as an NT service.)

The syntax of the `jmqsvcadmin` command is one of the following

```
jmqsvcadmin -h

jmqsvcadmin query

jmqsvcadmin remove

jmqsvcadmin install option [[option]...]
```

- The `query` subcommand displays the startup options to the `jmqsvcadmin` command. This includes whether the service is started manually or automatically, its location, the location of the java runtime, and the value of the arguments passed to the broker on startup.
- The `install` subcommand installs the service and specifies startup options.
- The `remove` subcommand removes the service.

Table C-1 `jmqsvcadmin` Options

Option	Description
`-h`	Displays help information for the `jmqsvcadmin` command syntax and options.
`-javahome` *path*	Specifies the path to an alternate Java 2 compatible JDK. The default is to use the bundled runtime. Example: `jmqsvcadmin -install -javahome d:\jdk1.3`
`-jrehome` *path*	Specifies the path to a Java 2 compatible JRE. Example: `jmqsvcadmin -install -jrehome d:\jre\1.3`
`-vmargs arg [[arg]...]`	Specifies additional arguments to pass to the Java VM that is running the broker service. (You can also specify these arguments in the NT Services Control Panel Startup Parameters field.) Example: `-vmargs "-Xms16m -Xmx128m"`

Table C-1 `jmqsvcadmin` Options *(Continued)*

Option	Description
`-args arg [[arg]...]`	Specifies additional command line arguments to pass to the broker service. For a description of the jmqbroker options, see "Starting a Broker" on page 109. (You can also specify these arguments in the NT Services Control Panel Startup Parameters field.) For example, `jmqsvcadmin -install -args "-passfile d:\jmqpassfile"`

The information that you specify using the `-javahome`, `-vmargs`, and `-args` options is stored in the Window's registry under the keys `JavaHome`, `JVMArgs`, and `ServiceArgs` in the path

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet
     \Services\JMQ_Broker\Parameters
```

Removing the Broker Service

Before you remove the broker service, you should use the `jmqcmd shutdown bkr` command to shut down the broker. Then use the `jmqsvcadmin remove` command to remove the service, and restart your computer.

Reconfiguring the Broker Service

To reconfigure the service, remove the service first, and then reinstall it, specifying different startup options with the `-args` argument.

Using an Alternate Java Runtime

You can use either the `-javahome` or `-jrehome` options to specify the location of an alternate java runtime. You can also specify these options in the NT Services Control Panel Startup Parameters field. Note that the Startup Parameters field treats the backslash (\) as an escape character, so you will have to type it twice when using it as a path delimiter; for example, `-javahome d:\\jdk1.3`.

Querying the Broker Service

To determine the startup options for the broker service, use the `-q` option to the `jmqsvcadmin` command.

```
jmqsvcadmin -query

Service JMQ_Broker is installed.
Display Name: JMQ_Broker
Start Type: Manual
Binary location: d:\jmq2.0\bin\jmqbrokersvc
JavaHome: d:\jdk1.3
Broker Args: -passfile d:\jmqpassfile
```

Troubleshooting

If you get an error when you try and start the service, you can see error events that were logged by doing the following.

➤ **To see logged service error events**

1. Start the Event Viewer
2. Look under Log > Application.
3. Select View > Refresh to see any error events.

Glossary

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

administered objects A pre-configured iMQ object—a connection factory or a destination—created by an administrator for use by one or more client applications.

The use of administered objects allows client applications to be provider-independent—that is, it isolates them from the proprietary aspects of a provider. These objects are placed in a JNDI name space by an administrator and are accessed by client applications using JNDI lookups.

asynchronous communication A mode of communication in which the sender of a message need not wait for the sending method to return before it continues with other work.

authorization The process by which a message service determines whether a user can access message service resources, such as connection services or destinations.

broker The iMQ entity that manages message routing, delivery, persistence, security, and logging, and which provides an interface that allows an administrator to monitor and tune performance and resource use.

client application An application (or software component) that interacts with other client applications using a message service to exchange messages.

client identifier An identifier that associates a connection and its objects with a state maintained by the iMQ Message Service on behalf of the client application.

client runtime See iMQ client runtime.

cluster Two or more interconnected brokers that work in tandem to provide messaging services.

configuration file One or more text files containing iMQ settings that are used to configure a broker. The properties are instance-specific or cluster-related.

connection 1) An active connection to an iMQ Message Service. This can be a queue connection or a topic connection. 2) A factory for sessions that use the connection underlying iMQ Message Service for producing and consuming messages.

connection factory The administered object the client uses to create a connection to iMQ Message Service. This can be a QueueConnectionFactory object or a TopicConnectionFactory object.

consume The receipt of a message taken from a destination by a message consumer.

consumer An object (MessageConsumer) created by a session that is used for receiving messages from a destination. In the point-to-point domain, the consumer is a receiver or browser (QueueReceiver or QueueBrowser); in the publish/subscribe domain, the consumer is a subscriber (TopicSubscriber).

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

delivery mode An indicator of the reliability of messaging: whether messages are guaranteed to be delivered and successfully consumed once and only once (persistent delivery mode) or guaranteed to be delivered at most once (non-persistent delivery mode)

delivery model The model by which messages are delivered: either point-to-point or publish/subscribe. In JMS there is a separate programming domain for each, using specific client runtime objects and specific destination types (queue or topic)

delivery policy A specification of how a queue is to route messages when more than one message consumer is registered. The policies are: single, failover, and round-robin.

destination The physical destination in an iMQ Message Service to which produced messages are delivered for routing and subsequent delivery to consumers. This physical destination is identified and encapsulated by an administered object that a client application uses to specify the destination of messages it is producing and the source of messages it is consuming.

domain A set of objects used by client applications to program JMS messaging operations. There are two programming domains: one for point-to-point messaging and one for publish/subscribe messaging.

iMQ client runtime Software that provides client applications with an interface to the iMQ Message Service. The client runtime supports all operations needed for clients to send messages to destinations and to receive messages from such destinations.

iMQ Message Service Software that provides delivery services for an iMQ messaging system, including connections to client applications, message routing and delivery, persistence, security, and logging. The Message Service maintains physical destinations to which client applications send messages, and from which the messages are delivered to consuming clients.

JMS (Java Message Service) A standard set of interfaces and semantics that define how a client application accesses the facilities of a message service. These interfaces provide a standard way for Java programs to create, send, receive, and read messages.

JMS provider A product that implements the JMS interfaces for a messaging system and adds the administrative and control functions needed for a complete product.

message selector A way for a consumer to select messages based on property values (selectors) in JMS message headers. A message service performs message filtering and routing based on criteria placed in message selectors.

message service See iMQ Message Service.

messages Asynchronous requests, reports, or events that are consumed by client applications. These messages contain information needed to coordinate these applications. A message has a header (to which additional fields can be added) and a body. The message header specifies standard fields and optional properties. The message body contains the data that is being transmitted.

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

point-to-point messaging A programming domain built around message queues. Producers address messages to specific queues; consumers extract messages from queues established to hold their messages.

produce Passing a message to the client runtime for delivery to a destination.

producer An object (MessageProducer) created by a session that is used for sending messages to a destination. In the point-to-point domain, a producer is a sender (QueueSender); in the publish/subscribe domain, a producer is a publisher (TopicPublisher).

publish/subscribe messaging A programming domain built around topics (in a content hierarchy). Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to a topic. The system distributes messages arriving from a topic's multiple publishers to its multiple subscribers.

queue An object created by an administrator to implement the point-to-point programming domain. A queue is always available to hold messages even when the client that consumes its messages is inactive. A queue is used as an intermediary holding place between producers and consumers.

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

topic An object created by an administrator to implement the publish/subscribe programming domain. A topic may be viewed as node in a content hierarchy that is responsible for gathering and distributing messages addressed to it. By using a topic as an intermediary, message publishers are kept separate from message subscribers.

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

user group The group to which the user of a client application belongs for purposes of authorizing access to iMQ Message Service resources, such as connections and destinations.

Index

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

U