Developer's Guide

iPlanet™ Message Queue for Java™

Version 2.0

June 5, 2001

Contents

List of Figures

List of Tables

List of Procedures

List of Code Examples

Preface

This book provides information regarding the concepts and procedures needed by a developer of messaging applications in an iPlanet Message Queue for Java (iMQ) environment.

This preface contains the following sections:

Audience for This Guide

This guide is meant principally for developers of messaging applications in an iMQ environment— applications that exchange messages using an iMQ Message Service.

These applications use the Java Message Service (JMS) Application Programming Interface (API) to create, send, receive, and read messages. The JMS specification is an open standard.

This *iMQ Developer's Guide* assumes that you are familiar with the JMS APIs and with JMS programming guidelines. It's purpose is to help you optimize your JMS client applications by making best use of the features and flexibility of an iMQ messaging system.

Organization of This Guide

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

Table 1 Book Contents

Chapter	Description
Chapter 1, "Overview"	A high level overview of the iMQ product and of JMS concepts and programming issues.
Chapter 2, "Quick Start Tutorial"	A tutorial that acquaints you with the iMQ development environment using a simple example client application.
Chapter 3, "Using Administered Objects"	Describes how to use iMQ administered objects in both a provider- independent and provider-specific way.
Chapter 4, "Optimizing Client Applications"	Explains features of the iMQ client runtime and how they can be used to optimize a client application.
Appendix A, "Administered Object Attributes"	Summarizes and documents administered object attributes.

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.

Table 2 Document Conventions *(Continued)*

Format	Description
`[]`	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.

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 Chapter 3, "Using Administered Objects"), which are of value to developers of messaging applications.

Example Client Applications

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

```
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
```

The specification includes sample client code.

Books on JMS Programming

For background on using the JMS API, you can consult the following publicly-available books:

- *Java Message Service* by Richard Monson-Haefel and David A. Chappell, O'Reilly and Associates, Inc., Sebastopol, CA
- *Professional JMS Programming* by Scott Grant, Michael P. Kovacs, Meeraj Kunnumpurath, Silvano Maffeis, K. Scott Morrison, Gopalan Suresh Raj, Paul Giotta, and James McGovern, Wrox Press Inc., ISBN: 1861004931
- *Practical Java Message Service* by Tarak Modi, Manning Publications, ISBN: 1930110138

Chapter 1

Overview

This chapter provides an overall introduction to the iMQ product and to JMS concepts and programming issues of interest to developers.

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.

With iPlanet Message Queue for Java software, processes running on different platforms and operating systems can connect to a common iMQ Message Service 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 messaging clients 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 implementing an 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 the *iMQ Administrator's Guide*). 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.

NOTE For all editions of iMQ, a portion of the product—the client APIs—can be freely redistributed for commercial use. All other files in the product *cannot* be redistributed. The portion that can be freely redistributed allows a licensee to develop an iMQ-based application (one which requires a connection to an iMQ Message Service) that they can sell to a third party without incurring any iMQ licensing fees. The third party will either need to purchase their own version of iMQ to access an iMQ Message Service or make a connection to yet another party that has an iMQ Message Service installed and running.

iMQ Messaging System Architecture

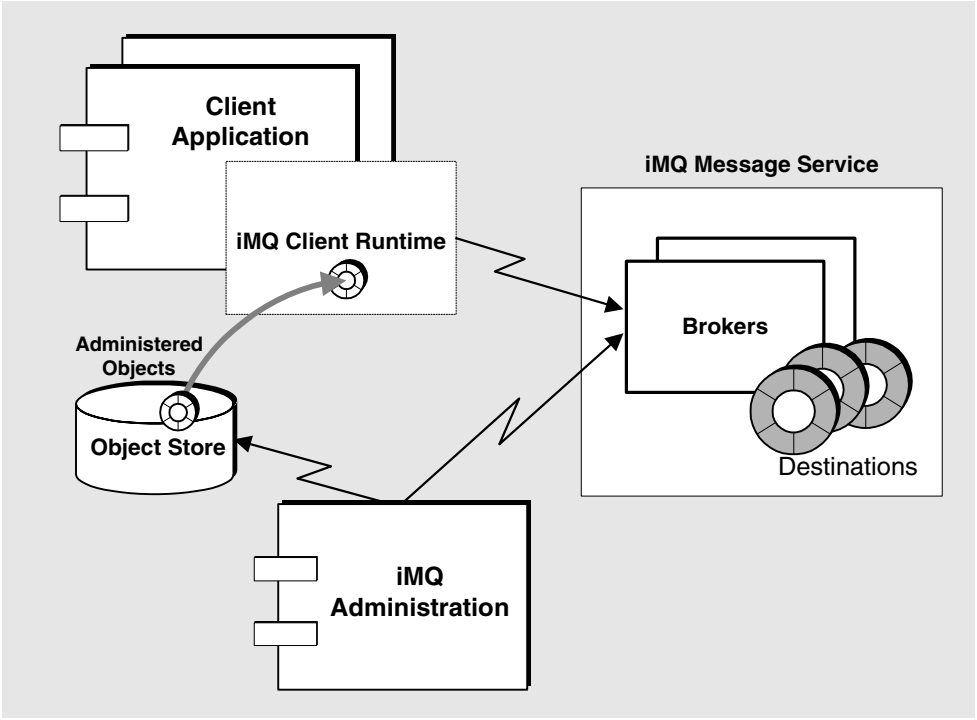
This section briefly describes the main parts of an iMQ messaging system. While as a developer, you do not need to be familiar with the details of all of these parts or how they interact, a high-level understanding of the basic architecture will help you understand features of the system that impact client application design and development.

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

iMQ Message Service The iMQ Message Service is the heart of a messaging system. It consists of one or more brokers which provide delivery services for the system. These services include 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. The iMQ Message Service is described in detail in the *iMQ Administrator's Guide*.

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 the JMS programming objects introduced in "The JMS Programming Model" on page 25. It supports all operations needed for clients to send messages to destinations and to receive messages from such destinations. The iMQ client runtime is described in detail in Chapter 4, "Optimizing Client Applications."

Figure 1-1 iMQ System Architecture



iMQ administered objects Administered Objects encapsulate provider-specific implementation and configuration information in objects that are used by client applications. Administered objects are generally 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. They can also be instantiated by client applications, in which case they are used in a provider-specific manner. Configuration of the iMQ client runtime is performed through administered object attributes, as described in Chapter 4, "Optimizing Client Applications."

iMQ administration iMQ provides a number of administration tools for managing an iMQ messaging system. These tools are used to manage the Message Service, create and store administered objects, manage security, manage messaging application resources, and manage persistent data. These tools are generally used by iMQ administrators and are described in the *iMQ Administrator's Guide*.

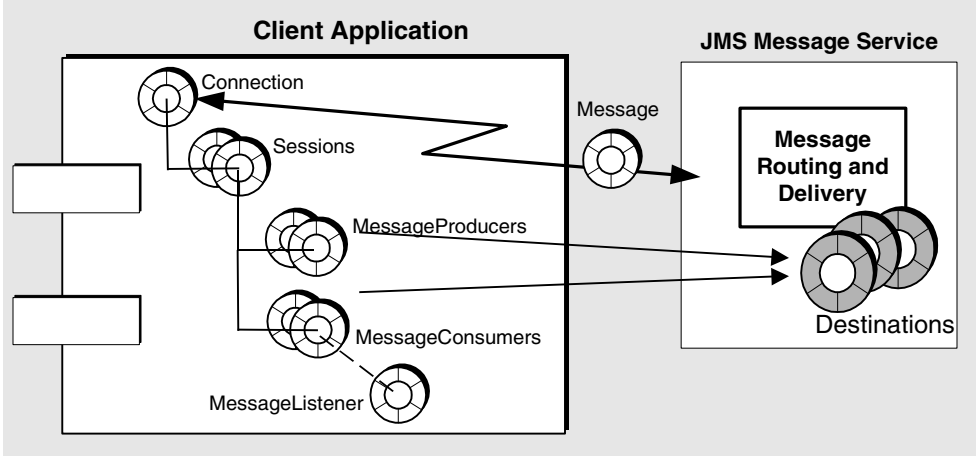
The JMS Programming Model

This section briefly describes the programming model of the JMS specification. It is meant as a review of the most important concepts and terminology used in programming JMS client applications.

JMS Programming Interface

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 (see "Client Setup Operations" on page 29). The main interface objects are shown in Figure 1-2 and described in the following paragraphs.

Figure 1-2 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 clients can use to filter messages. A body is also optional—it contains the actual data to be exchanged.

Header

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

Some header field values are set automatically by iMQ during the process of producing and delivering a message, some depend on settings of message producers specified when the message producers are created in the client application, and others are set on a message by message basis by the client application using JMS APIs. The following table lists the header fields defined (and required) by JMS, as well as how they are set.

Table 1-1 JMS-defined Message Header

Header Field	Set By:	Default
`JMSDestination`	Client application, for each message producer or message	
`JMSDeliveryMode`	Client application, for each message producer or message	`Persistent`
`JMSExpiration`	Client application, for each message producer or message	time to live is `0` (no expiration)
`JMSPriority`	Client application, for each message producer	`4` (normal)
`JMSMessageID`	Provider, automatically	
`JMSTimestamp`	Provider, automatically	
`JMSRedelivered`	Provider, automatically	
`JMSCorrelationID`	Client application, for each message	
`JMSReplyTo`	Client application, for each message	
`JMSType`	Client application, for each message	

Properties

When data is sent between two processes, other information besides the payload data can be sent with it. These descriptive fields, or properties, can provide additional information about the data, including which process created it, the time it was created, and information that uniquely identifies the structure of each piece of data. Properties consist of property name and property value pairs, as specified by a client application.

Having registered an interest in a particular destination, consuming clients can fine-tune their selection by specifying certain property values as selection criteria. For instance, a client might indicate an interest in Payroll messages (rather than Facilities) but only Payroll items concerning part-time employees located in New Jersey. Messages that do not meet the specified criteria are not delivered to the consumer.

Message Body Types

JMS specifies six classes (or types) of messages that a JMS provider must support, as described in the following table:

Table 1-2 Message Body Types

Type	Description
Message	a message without a message body.
StreamMessage	a message whose body contains a stream of Java primitive values. It is filled and read sequentially.
MapMessage	a message whose body contains a set of name-value pairs. The order of entries is not defined.
TextMessage	a message whose body contains a Java string, for example an XML message.
ObjectMessage	a message whose body contains a serialized Java object.
BytesMessage	a message whose body contains a stream of uninterpreted bytes.

Destination

A `Destination` is a JMS administered object (see Chapter 3, "Using Administered Objects") 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 Chapter 3, "Using Administered Objects") 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 "Message Consumption: Synchronous and Asynchronous" on page 35).

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.

Client Setup Operations

There is a general approach within the JMS programming model for setting up a client to produce or consume messages. It uses the JMS programming interface objects described in the previous section.

The general procedures for producing and consuming messages are introduced below. The procedures have a number of common steps which need not be duplicated if a client application is both producing and consuming messages.

➤ **To set up a client application to produce messages**

1. Use JNDI to find a `ConnectionFactory` object. (You can also directly instantiate a `ConnectionFactory` object and set its attribute values.)
2. Use the `ConnectionFactory` object to create a `Connection` object.
3. Use the `Connection` object to create one or more `Session` objects.
4. Use JNDI to find one or more `Destination` objects. (You can also directly instantiate a `Destination` object and set its name attribute.)
5. Use a `Session` object and a `Destination` object to create any needed `MessageProducer` objects. (You can create a `MessageProducer` object without specifying a `Destination` object, but then you have to specify a `Destination` object for each message that you produce.)

At this point the client application has the basic setup needed to produce messages.

➤ **To set up a client application to consume messages**

1. Use JNDI to find a `ConnectionFactory` object. (You can also directly instantiate a `ConnectionFactory` object and set its attribute values.)
2. Use the `ConnectionFactory` object to create a `Connection` object.
3. Use the `Connection` object to create one or more `Session` objects.
4. Use JNDI to find one or more `Destination` objects. (You can also directly instantiate a `Destination` object and set its name attribute.)
5. Use a `Session` object and a `Destination` object to create any needed `MessageConsumer` objects.
6. If needed, instantiate a `MessageListener` object and register it with a `MessageConsumer` object.
7. Tell the `Connection` object to start delivery of messages. This allows messages to be delivered to the client for consumption.

At this point the client application has the basic setup needed to consume messages.

JMS Application Design Issues

This section is a review of a number of JMS messaging issues that impact application design.

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 the *iMQ Administrator's Guide*), 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 1-3. 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 1-3 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 Chapter 3, "Using Administered Objects"). 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 31) 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 (see "Client Identification" on page 61). 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.

By default, if no other means of setting the client identifier has been chosen, the iMQ client runtime (see "iMQ client runtime" on page 23) 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. 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 persistence and acknowledgement properties (see "Performance Factors" on page 67).

Message Consumption: Synchronous and Asynchronous

There are two ways a client application can consume messages: either synchronously or asynchronously.

In synchronous consumption, a client gets a message by invoking the `receive()` method of a `MessageConsumer` object. The client thread blocks until the method returns. This means that if no message is available, the client blocks until a message does become available or until the `receive()` method times out (if it was called with a time-out specified). In this model, a client thread can only consume messages one at a time (synchronously).

In asynchronous consumption, a client registers a `MessageListener` object with a message consumer. The message listener is like a call-back object. A client consumes a message when the session invokes the `onMessage()` method of the `MessageListener` object. In this model, the client thread does not block (message is asynchronously consumed) because the thread listening for and consuming the message belongs to the iMQ client runtime.

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 19).

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 a number of 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 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 the *iMQ Administrator's Guide*), and message service availability.

Chapter 2

Quick Start Tutorial

This chapter provides a quick introduction to JMS client application programming in an iMQ environment. It consists of a tutorial-style description of procedures used to create, compile, and run a simple HelloWorldMessage example application.

This chapter covers the following procedures:

- setting up your environment
- starting and testing a broker
- developing a simple client application
- compiling and running a client application

For the purpose of this tutorial it is sufficient to run the iMQ Message Service in a default configuration. For instructions on configuring an iMQ Message Service, please refer to the *iMQ Administrator's Guide*.

Setting Up Your Environment

A number of environment variables are used when compiling and running a JMS client application. This section describes how to set them.

➤ **To set iMQ-related environment variables**

1. Set the `JMQ_HOME` environment variable.

   The `JMQ_HOME` environment variable is used when compiling and running a client application. On some platforms, the iMQ installer sets `JMQ_HOME`.

The directories and environment variables that are set up at installation time are listed below.

Platform	Settings
On Solaris and Linux	The default installation directory is `/opt/SUNWjmq`
	The `JMQ_HOME` environment variable is not set by the installer.
On Windows	The default installation directory is `C:\Program Files\iPlanetMessageQueue2.0\`
	The `JMQ_HOME` environment variable is set by the installer to `C:\Program Files\iPlanetMessageQueue2.0\`

Because `JMQ_HOME` is not set by the Solaris and Linux installer, you have to set it on those platforms by hand:

```
% setenv JMQ_HOME /opt/SUNWjmq
```

2. Set the `JAVA_HOME` environment variable.

   Set `JAVA_HOME` to the directory where you installed the J2SE SDK (Java2 Standard Edition Software Development Kit).

3. Change directory to `JMQ_HOME/examples/jms`.

   This is the directory in which you will find a number of example applications, including the one used as an example in this chapter.

4. Set the `CLASSPATH` environment variable.

   Set `CLASSPATH` to include the current `JMQ_HOME/examples/jms` directory, as well as the jar files (`jms.jar, jmq.jar, jndi.jar`) in the `JMQ_HOME/lib` directory that are required for compiling and running client applications.

Platform	Settings and Details

On Solaris and Linux (csh)	`% cd $JMQ_HOME/examples/jms`
	`% setenv CLASSPATH .:$JMQ_HOME/lib/jms.jar:`
	`$JMQ_HOME/lib/jmq.jar:$JMQ_HOME/lib/jndi.jar`

On Windows	`C:\>cd %JMQ_HOME%\examples\jms`
	`C:\Program Files\JavaMessageQueue2.0\examples\jms>`
	`set CLASSPATH=.;%JMQ_HOME%\lib\jms.jar;`
	`%JMQ_HOME%\lib\jmq.jar;%JMQ_HOME%\lib\jndi.jar`

Starting and Testing the iMQ Message Service

This tutorial assumes that you do not have an iMQ Message Service currently running. A Message Service consists of one or more brokers—the software component that routes and delivers messages.

(If, during installation you chose to run the broker as a UNIX background process or Windows service, then it is already running and you can skip to "To test a broker" below.)

➤ To start a broker

1. In a terminal window, change directory to `JMQ_HOME/bin`.
2. Run the iMQ Broker (`jmqbroker`) command as described below.

   The `-tty` option causes all logged messages to be displayed to the terminal console (in addition to the log file).

Platform	Startup Command
On Solaris and Linux	`% ./jmqbroker -tty`
On Windows	`C:\Program Files\iPlanetMessageQueue2.0\bin>` `jmqbroker -tty`

The broker will start and display a few messages before displaying the message, "Broker ready." It is now ready and available for client applications to use.

➤ To test a broker

One simple way to check the broker startup is by using the iMQ Command (`jmqcmd`) utility to display information about the broker.

1. In a separate terminal window, change directory to `JMQ_HOME/bin`.
2. Run `jmqcmd` with the arguments shown below.

Platform	Settings and Details
On Solaris and Linux	`% ./jmqcmd query bkr -u admin -p admin`
On Windows	`C:\Program Files\iPlanetMessageQueue2.0\bin>` `jmqcmd query bkr -u admin -p admin`

The output displayed should be similar to what is shown below.

```
Querying the broker specified by:
-------------------------
Host           Primary Port
-------------------------
localhost       7676

Instance Name                          jmqbroker
Primary Port                           7676
Auto Create Topics                     true
Auto Create Queues                     true
Log Level                              INFO
Log Rollover Size (bytes)              0 (unlimited)
Log Rollover Interval (seconds)        604800
Swap Threshold: Number of Messages     0 (unlimited)
Swap Threshold: Size of Messages       70
Max Number of Messages in System       0 (unlimited)
Max Size of Messages in System         0 (unlimited)
Max Message Size (bytes)               70
Cluster Broker List (configured)       myhost/111.222.333.444:7676 jmqbroker
Cluster Broker List (active)
Cluster Master Broker
Cluster URL

Successfully queried the broker.
```

Developing a Simple Client Application

This section leads you through the steps used to create a simple "Hello World" client application that sends a message to a queue destination and then retrieves the same message from the queue. You can find this HelloWorldMessage application at `JMQ_HOME/examples/jms`.

The following steps highlight Java programming language code that you use to set up a client to send and receive messages:

➤ **To program the HelloWorldMessage example application**

1. Import the interfaces and iMQ implementation classes for the JMS API.

   The javax.jms package defines all the JMS interfaces necessary to develop a JMS client application.

   ```
   import javax.jms.*;
   ```

2. Instantiate an iMQ `QueueConnectionFactory` administered object.

   A `QueueConnectionFactory` object encapsulates all the iMQ-specific configuration properties for creating `QueueConnection` connections to an iMQ Message Service.

   ```
   QueueConnectionFactory myQConnFactory =
       new com.sun.messaging.QueueConnectionFactory();
   ```

   `ConnectionFactory` administered objects can also be accessed through a JNDI lookup (see "Looking Up ConnectionFactory Objects" on page 49). This approach makes the client code JMS-provider independent and also allows for a centrally administered messaging system.

3. Create a connection to the iMQ Message Service.

   A `QueueConnection` object is the active connection to the iMQ Message Service in the Point-To-Point programming domain.

   ```
   QueueConnection myQConn =
       myQConnFactory.createQueueConnection();
   ```

4. Create a session within the connection.

   A `QueueSession` object is a single-threaded context for producing and consuming messages. It enables clients to create producers and consumers of messages for a queue destination.

   ```
   QueueSession myQSess = myQConn.createQueueSession(false,
       Session.AUTO_ACKNOWLEDGE);
   ```

   The `myQSess` object created above is non-transacted and automatically acknowledges messages upon consumption by a consumer.

5. Instantiate an iMQ `Destination` administered object corresponding to a queue destination in the iMQ Message Service.

   `Destination` administered objects encapsulate provider-specific destination naming syntax and behavior. The code below instantiates a `Destination` administered object for a physical queue destination named "world".

   ```
   Queue myQueue = new.com.sun.messaging.Queue("world");
   ```

   `Destination` administered objects can also be accessed through a JNDI lookup (see "Looking Up Destination Objects" on page 50). This approach makes the client code JMS-provider independent and also allows for a centrally administered messaging system.

6. Create a `QueueSender` message producer.

   This message producer, associated with `myQueue`, is used to send messages to the queue destination named "world".

   ```
   QueueSender myQueueSender = myQSess.createSender(myQueue);
   ```

7. Create and send a message to the queue.

   You create a `TextMessage` object using the `QueueSession` object and populate it with a string representing the data of the message. Then you use the `QueueSender` object to send the message to the "world" queue destination.

   ```
   TextMessage myTextMsg = myQSess.createTextMessage();
   myTextMsg.setText("Hello World");
   myQueueSender.send(myTextMsg);
   ```

8. Create a `QueueReceiver` message consumer.

   This message consumer, associated with `myQueue`, is used to receive messages from the queue destination named "world".

   ```
   QueueReceiver myQueueReceiver =
      myQSess.createReceiver(myQueue);
   ```

9. Start the `QueueConnection` you created in Step 3.

   Messages for consumption by a client can only be delivered over a connection that has been started (while messages produced by a client can be delivered to a destination without starting a connection, as in Step 7.

   ```
   myQConn.start();
   ```

10. Receive a message from the queue

    You receive a message from the "world" queue destination using the `QueueReceiver` object. The code, below, is an example of a synchronous consumption of messages (see "Message Consumption: Synchronous and Asynchronous" on page 35). For samples of asynchronous consumption see Table 2-1 on page 44.

    ```
    Message msg = myQueueReceiver.receive();
    ```

11. Retrieve the contents of the message.

    Once the message is received successfully, it's contents can be retrieved.

```
if (msg instanceof TextMessage) {
    TextMessage txtMsg = (TextMessage) msg;
    System.out.println("Read Message: " + txtMsg.getText());
}
```

12. Close the session and connection resources.

```
myQSess.close();
myQConn.close();
```

Compiling and Running a Client Application

This section assumes that you are using the Java2 SDK Standard Edition v1.3, the recommended version (1.2.2 and 1.1.8 are also supported), to compile and run JMS client applications in an iMQ environment. This SDK can be downloaded from:

```
http://java.sun.com/j2se/1.3
```

Be sure that you have set the CLASSPATH environment variable to point to the `jms.jar`, `jmq.jar`, and `jndi.jar` files, as described in Step 4 on page 38, before attempting to compile or run a client application.

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

1. Compile the HelloWorldMessage application as shown below.

Platform	Settings and Details
On Solaris and Linux (csh)	`% $JAVA_HOME/bin/javac HelloWorldMessage.java`
On Windows	`C:\Program Files\iPlanetMessageQueue2.0\examples\jms>` `%JAVA_HOME%\bin\javac HelloWorldMessage.java`

This step above results in the `HelloWorldMessage.class` file being created in the `JMQ_HOME/examples/jms` directory.

2. Run the HelloWorldMessage application as shown below.

   The output shown is what displays when you run the HelloWorldMessage example in the examples/jms directory.

Platform	Settings and Details

On Solaris and Linux (csh)	`% $JAVA_HOME\bin\java HelloWorldMessage`
	
	`Sending Message: Hello World`
	`Read Message: Hello World`

On Window	`C:\Program Files\JavaMessageQueue2.0\examples\jms>`
	`%JAVA_HOME%\bin\java HelloWorldMessage`
	
	`Sending Message: Hello World`
	`Read Message: Hello World`

Example Application Code

A listing of the code in the HelloWorldMessage tutorial example can be found, along with code from a number of other example applications, at the following location:

```
JMQ_HOME/examples/jms
```

The directory includes a `README` file that describes each example application and how to run it. The examples include standard JMS sample programs as well as iMQ-supplied example applications. They are summarized in the following two tables.

Table 2-1 is a listing and brief description of the JMS sample programs.

Table 2-1 JMS Sample Programs

Name of Example Application	Description
SenderToQueue	Sends a text message using a queue.
SynchQueueReceiver	Synchronously receives a text message using a queue.
SynchTopicExample	Publishes and synchronously receives a text message using a topic.
AsynchQueueReceiver	Asynchronously receives a number of text messages using a message listener.

Table 2-1 JMS Sample Programs *(Continued)*

Name of Example Application	Description
AsynchTopicExample	Publishes five text messages to a topic and asynchronously gets them using a message listener.
MessageFormats	Writes and reads messages in five supported message formats.
MessageConversion	Shows that for some message formats, you can write a message using one data type and read it using another.
ObjectMessages	Shows that objects are copied into messages, not passed by reference.
BytesMessages	Shows how to write, then read, a Bytes Message of indeterminate length.
MessageHeadersTopic	Illustrates the use of the JMS message header fields.
TopicSelectors	Shows how to use message properties as message selectors.
DurableSubscriberExample	Shows how you can create a durable subscriber that retains messages published to a topic while the subscriber is inactive.
AckEquivExample	Shows how to ensure that a message will not be acknowledged until processing is complete.
TransactedExample	Demonstrates the use of transactions in a simulated e-commerce application.
RequestReplyQueue	Demonstrates use of the JMS request/reply facility.

Table 2-2 is a listing and brief description of the iMQ-supplied example applications.

Table 2-2 iMQ-supplied Example Applications

Name of Example Application	Description
HelloWorldMessage	Sends and receives a “Hello World” message.
XMLMessageExample	Reads an XML document from a file, sends it to a queue, processes the message from the queue as an XML document, and converts it to a DOM object.
SimpleChat	Illustrates how iMQ can be used to create a simple GUI chat application.

Chapter 3

Using Administered Objects

Administered Objects encapsulate provider-specific implementation and configuration information in objects that are used by client applications.

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 create message consumers and producers (see "Developing a Simple Client Application" on page 40.)

There are two approaches to the use of Administered objects:

- They can be 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.
- They can be instantiated and configured by a developer when writing client application code. In this case, they are used in a provider-specific manner.

The approach you take in using iMQ administered objects depends on the environment in which your application will be run and how much control you want your client application to have over iMQ-specific configuration details. This chapter describes these two approaches and explains how to code your client applications for each.

JNDI Lookup of Administered Objects

If you wish an application to be run under controlled conditions in a centrally administered messaging environment, then iMQ administered objects should be created and configured by an administrator. This makes it possible for the administrator to:

- control the behavior of connections by requiring client applications to access pre-configured `ConnectionFactory` objects through a JNDI lookup.
- control the proliferation of physical destinations by requiring client applications to access only `Destination` objects that correspond to existing physical destinations.

This approach gives the administrator control over Message Service and client runtime configuration details, and at the same time allows client applications to be JMS provider-independent: they do not have to know about provider-specific syntax and object naming conventions or provider-specific configuration properties.

An administrator creates administered objects using iMQ administration tools, as described in the *iMQ Administrator's Guide*. When creating an administered object, the administrator can specify that it be read only—that is, client applications cannot change iMQ-specific configuration values specified when the object was created. In other words, client code cannot set attribute values on read-only administered objects, nor can they be overridden using client application startup options, as described in "Starting Client Applications With Overrides" on page 54.

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 an administrator to control the broker resources required by an application and to tune application performance. Instantiating administered objects also makes client applications provider specific.

Looking Up ConnectionFactory Objects

➤ **To perform a JNDI lookup of a ConnectionFactory object**

1. Create an initial context for the JNDI lookup.

   The details of how you create this context depend on whether you are using a file-system store or an LDAP server for your iMQ administered objects. The code below assumes a file-system store. For information about the corresponding LDAP server properties, see the *iMQ Administrator's Guide*.

   ```
   Hashtable env = new Hashtable();
   env.put (Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.fscontext.RefFSContextFactory");
   env.put (Context.PROVIDER_URL,
      "file:///c:/jmq_admin_objects");
   Context ctx = new InitialContext(env);
   ```

   You can also set an environment by specifying system properties on the command line, rather than programmatically, as shown above. For instructions, see the `README` file in the example applications directory:

   ```
   JMQ_HOME/examples/jms
   ```

   If you use system properties to set the environment, then you initialize the context without providing the `env` parameter:

   ```
   Context ctx = new InitialContext();
   ```

2. Perform a JNDI lookup on the "lookup" name under which the `ConnectionFactory` object was stored in the JNDI object store.

   ```
   QueueConnectionFactory myQConnFactory = (QueueConnectionFactory)
      ctx.lookup("cn=MyQueueConnectionFactory");
   ```

   It is recommended that you use this connection factory as originally configured. For a discussion of `ConnectionFactory` configuration properties, see "iMQ Client Runtime Configurable Properties" on page 59 and for a complete list of properties, see "ConnectionFactory Administered Object" on page 69.

3. Use the ConnectionFactory to create a connection object.

```
QueueConnection myQConn =
    myQConnFactory.createQueueConnection();
```

The code in the previous steps is shown in Code Example 3-1.

Code Example 3-1 Looking Up a ConnectionFactory Object

```
Hashtable env = new Hashtable();
env.put (Context.INITIAL_CONTEXT_FACTORY,
  "com.sun.jndi.fscontext.RefFSContextFactory");
env.put (Context.PROVIDER_URL,
  "file:///c:/jmq_admin_objects");
Context ctx = new InitialContext(env);
QueueConnectionFactory myQConnFactory = (QueueConnectionFactory)
    ctx.lookup("cn=MyQueueConnectionFactory");
QueueConnection myQConn =
    myQConnFactory.createQueueConnection();
```

Looking Up Destination Objects

➤ **To perform a JNDI lookup of a Destination object**

1. Using the same initial context used in performing the `ConnectionFactory` lookup, Perform a JNDI lookup on the "lookup" name under which the `Destination` object was stored in the JNDI object store.

```
Queue myQ =
(Queue) ctx.lookup("cn=MyQueueDestination");
```

Instantiating Administered Objects

If you do not wish an application to be run under controlled conditions in a centrally administered environment, then you can instantiate and configure administered objects in client application code.

While this approach gives you, the developer, control over Message Service and client runtime configuration details, it also means that your client applications are not supported by other JMS providers. Typically, you might instantiate administered objects in client code in the following situations:

- You are in the early stages of development in which there is no real need to create, configure, and store administered objects. You just want to develop and debug your application without involving JNDI lookups.
- You are not concerned about your client applications being supported by other JMS providers.

Instantiating administered objects in client code means you are hard-coding configuration values into your application. You give up the flexibility of having an administrator reconfigure the administered objects to achieve higher performance or throughput after an application has been deployed.

Instantiating ConnectionFactory Objects

There are two object constructors for instantiating iMQ `ConnectionFactory` administered objects, one for each programming domain:

- **Publish/subscribe (Topic) domain**

  ```
  new com.sun.messaging.TopicConnectionFactory();
  ```

  Instantiates a `TopicConnectionFactory` with a default configuration (creates Topic TCP-based connections to a broker running on "`localhost`" at port number `7676`).

- **Point to point (Queue) domain**

  ```
  new com.sun.messaging.QueueConnectionFactory();
  ```

  Instantiates a `QueueConnectionFactory` with a default configuration (creates Queue TCP-based connections to a broker running on "`localhost`" at port number `7676`).

➤ **To directly instantiate and configure a ConnectionFactory object**

1. Instantiate a Topic or Queue `ConnectionFactory` object using the appropriate constructor.

   ```
   com.sun.messaging.QueueConnectionFactory myQConnFactory =
       new com.sun.messaging.QueueConnectionFactory();
   ```

2. Configure the `ConnectionFactory` object.

   ```
   myQConnFactory.setProperty("JMQBrokerHostName", "new_hostname");
   myQConnFactory.setProperty("JMQBrokerHostPort", "7878");
   ```

   For a discussion of `ConnectionFactory` configuration properties, see "iMQ Client Runtime Configurable Properties" on page 59 and for a complete list of properties, see "ConnectionFactory Administered Object" on page 69.

3. Use the ConnectionFactory to create a `Connection` object.

   ```
   QueueConnection myQConn =
       myQConnFactory.createQueueConnection();
   ```

The code in the previous steps is shown in Code Example 3-2.

Code Example 3-2 Instantiating a ConnectionFactory Object

```
com.sun.messaging.QueueConnectionFactory myQConnFactory =
    new com.sun.messaging.QueueConnectionFactory();
try {
    myQConnFactory.setProperty("JMQBrokerHostName", "new_host");
    myQConnFactory.setProperty("JMQBrokerHostPort", "7878");
} catch (JMSException je) {
}
QueueConnection myQConn =
    myQConnFactory.createQueueConnection();
```

Instantiating Destination Objects

There are two object constructors for instantiating iMQ `Destination` administered objects, one for each programming domain:

- **Publish/subscribe (Topic) domain**

  ```
  new com.sun.messaging.Topic();
  ```

  Instantiates a `Topic` with the default destination name of "Untitled_Destination_Object".

- **Point to point (Queue) domain**

  ```
  new com.sun.messaging.Queue();
  ```

  Instantiates a `Queue` with the default destination name of "Untitled_Destination_Object".

➤ **To directly instantiate and configure a Destination object**

1. Instantiate a Topic or Queue `Destination` object using the appropriate constructor.

   ```
   com.sun.messaging.Queue myQueue = new com.sun.messaging.Queue();
   ```

2. Configure the Destination object.

   ```
   myQueue.setProperty("JMQDestinationName", "new_queue_name");
   ```

3. After creating a session, you use the `Destination` object to create a MessageProducer or MessageConsumer object.

   ```
   QueueSender qs = qSession.createSender((Queue)myQueue);
   ```

The code is shown in Code Example 3-3.

Code Example 3-3 Instantiating a Destination Object

```
com.sun.messaging.Queue myQueue = new com.sun.messaging.Queue();
try {
    myQueue.setProperty("JMQDestinationName", "new_queue_name");
} catch (JMSException je) {
}
...
QueueSender qs = qSession.createSender((Queue)myQueue);
...
```

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. You can override the configuration of iMQ administered objects accessed through a JNDI lookup as well as iMQ administered objects instantiated and configured using `setProperty()` methods in client application code.

To override administered object settings, use the following command line syntax:

`java [[-D`*attribute*`=`*value* `]...]` *clientAppName*

where `attribute` corresponds to any of the `ConnectionFactory` administered object attributes documented in "iMQ Client Runtime Configurable Properties" on page 59.

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.

It is also possible to set system properties within client code using the `System.setProperty()` method. This method will override attribute values of iMQ administered objects in the same way that command line options do.

If an administered object has been set as read-only, however, the values of its attributes cannot be changed using either command-line overrides or the `System.setProperty()` method. Any such overrides will simply be ignored.

Chapter 4

Optimizing Client Applications

The performance of client applications depends both on the inherent design of these applications and on the features and capabilities of the iMQ client runtime.

This chapter describes how the iMQ client runtime supports the messaging capabilities of client applications, with special emphasis on properties and behaviors that can be configured to improve performance and message throughput.

The chapter covers the following topics:

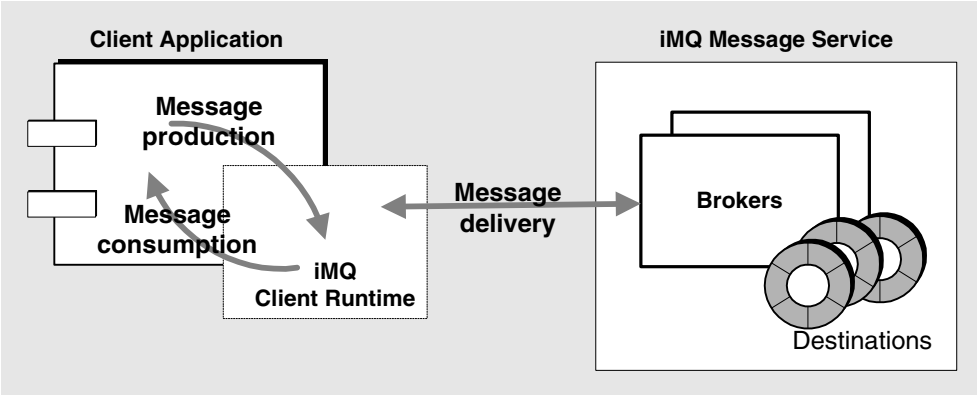
- message production and consumption
- configurable properties of the iMQ client runtime
- factors that affect performance

Message Production and Consumption

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 25. 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 supports message production and consumption. Figure 4-1 on page 56 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 4-1 Messaging Operations



Once a client application has created a connection to a broker, created a session as a single-threaded context for message delivery, and created the MessageProducer and MessageConsumer objects needed to access particular destinations in a Message Service, production (sending) and consumption (receiving) of messages can proceed.

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.

In the case of persistent messages, to increase throughput, you can set the connection to *not* require broker acknowledgement (see `JMQAckOnProduce` property, Table 4-6 on page 65), but this eliminates the guarantee that persistent messages are reliably delivered.

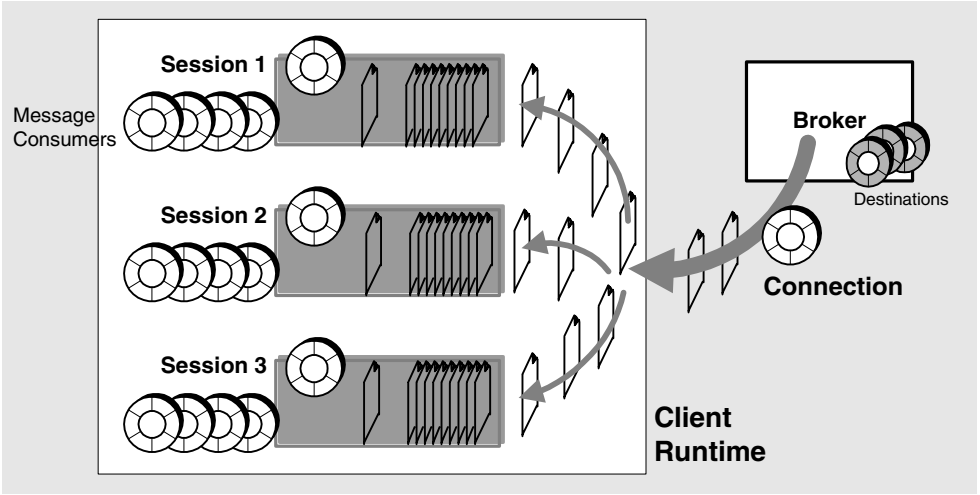
Message Consumption

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 4-2. 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 4-2 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.

In accordance with the JMS specification, there are three acknowledgment options that a client developer can set for a client session:

- `AUTO_ACKNOWLEDGE`: the session automatically acknowledges each message consumed by the client.
- `CLIENT_ACKNOWLEDGE`: the client application explicitly acknowledges after one or more messages have been consumed. This option gives the client the most control.
- `DUPS_OK_ACKNOWLEDGE`: the session acknowledges after a configurable number of messages have been consumed and doesn't guarantee that messages are delivered and consumed only once. Client applications use this mode if they don't care if messages are processed more than once.

Each of the three acknowledgement options requires a different level of processing and bandwidth overhead. Automatic acknowledge consumes the most overhead and guarantees reliability on a message by message basis, while `DUPS_OK_ACKNOWLEDGE` consumes the least overhead, but allows for duplicate delivery of messages.

In the case of `AUTO_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE` acknowledgement, the threads performing the acknowledgement, or committing a transaction, will block, waiting for the broker to return a control message acknowledging receipt of the client acknowledgement. This broker acknowledgement (referred to as "Ack" in property names) guarantees that the broker has deleted the corresponding persistent message and will not send it twice—which could happen were the client or broker to fail, or the connection to fail, at the wrong time.

To increase throughput, you can set the connection to *not* require broker acknowledgement of client acknowledgements (see `JMQAckOnAcknowledge` property, Table 4-4 on page 63), but this eliminates the guarantee that persistent messages are reliably delivered.

NOTE In the `DUPS_OK_ACKNOWLEDGE` mode, the session does not wait for broker acknowledgements. This option is used in client applications in which duplicate messages are not a problem. Also, in the `CLIENT_ACKNOWLEDGE` mode, there is a JMS API (`recover Session`) by which a client can explicitly request redelivery of messages that have been received but not yet acknowledged by the client. When redelivering such messages, the broker marks them with a Redeliver flag.

iMQ Client Runtime Configurable Properties

The iMQ client runtime supports all the operations described in "Message Production and Consumption" on page 55. It also provides a number of configurable properties that you can use to optimize performance and message throughput. These properties correspond to attributes of the `ConnectionFactory` object 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. The `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.

`ConnectionFactory` administered objects are created by adminstrators or instantiated in client code, as described in Chapter 3, "Using Administered Objects."

`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 is discussed in the following sections with a description of the `ConnectionFactory` attributes each includes.

Connection Specification

Connections are specified by a broker's host name, the port number at which its Port Mapper resides, and the kind of connection service it supports. The behavior of a connection might require setting additional attribute values, depending on the connection type (the protocol used by the connection service).

Table 4-1 Connection Factory Attributes: Connection Handling

Attribute/property name	Description
`JMQConnectionType`	Specifies transport protocol used by connection service. Supported types are `TCP`, `SSL`, `HTTP`. Default: `TCP`
`JMQAckTimeout`	Specifies the maximum time in milliseconds that the client runtime will wait for any broker acknowledgement before throwing an exception. A value of 0 means there is no time-out. Default: `0`
`JMQBrokerHostName`	Specifies the broker host name to which to connect (if `JMQConnectionType` is either `TCP` or `SSL`). Default: `localhost`
`JMQBrokerHostPort`	Specifies the broker host port (if `JMQConnectionType` is either `TCP` or `SSL`). Default: `7676`
`JMQSSLIsHostTrusted`	Specifies whether the host is trusted (if `JMQConnectionType` is `SSL`). Default: `true`
`JMQConnectionURL`	Specifies the URL that will be used to connect to the iMQ Message Service (if `JMQConnectionType` is `HTTP`). Default: `http://localhost/servlet/HttpTunnelServlet`

Auto-reconnect Behavior

These attributes configure iMQ's automatic reconnect capability. If a connection fails, iMQ maintains objects provided by the client runtime (sessions, message consumers, message producers, and so forth) while attempting to re-establish the connection. Producers cannot send during reconnect (they continue to retry until the connection is re-established). For consumers, however, messages that have been delivered to the client runtime will continue to be available for consumption while auto-reconnect is being attempted.

Table 4-2 Connection Factory Attributes: Auto-reconnect Behavior

Attribute/property name	Description
`JMQReconnect`	Specifies whether the client runtime will attempt to reconnect to the broker if the connection is lost. Default: `false`
`JMQReconnectDelay`	Specifies the time between successive attempts of the client runtime to reconnect to the iMQ Message Service (if `JMQReconnect=true`). Default: `30000 milliseconds`
`JMQReconnectRetries`	Specifies the number of attempts the client runtime will make to reconnect to the broker (if `JMQReconnect=true`). A value of 0 indicates that the number of retries is not limited. Default: `0`

Client Identification

These attributes specify default values of user name and password and specify how a client will be identified (ClientID) to a broker.

The default user name and password is provided as a convenience for developers (see the *iMQ Administrator's Guide*).

ClientID is used principally to keep track of durable subscriptions (see "Publish/Subscribe—Topic destinations" on page 31). If a durable subscriber is inactive at the time that messages are delivered to a topic destination, the broker retains messages for that subscriber and delivers them when the subscriber once again becomes active. The only way for the broker to identify the subscriber is through the use of a unique ClientID.

There are a number of ways that the ClientID can be set for a connection. For example, client application code can use the setClientID() method of a `Connection` object. The ClientID must be set before using the connection in any way; once the connection is used, the ClientID cannot be set or reset.

Setting the ClientID in client application code, however, is not optimal. Each user needs a unique identification: this implies some centralized coordination. iMQ therefore provides a `JMQConfiguredClientID` attribute on the ConnectionFactory object. This attribute can be used to provide a unique ClientID to each user. To use this feature, the value of `JMQConfiguredClientID` is set as follows:

`JMQConfiguredClientID=${u}`*string*

where the special reserved characters, ${u}, provide a unique user identification during the user authentication stage of establishing a connection, and *string* is a text value unique to the ConnectionFactory object. When used properly, the iMQ Message Service will substitute `u:`*username* for the u, resulting in a user-specific ClientID.

The `${u}` must be the first four characters of the attribute value. If anything other than "u" is encountered, it will result in an JMS exception upon connection creation. When `${}` is used anywhere else in the attribute value, it is treated as plain text and no variable substitution is performed.

An additional attribute, `JMQDisableSetClientID`, can be set to `true` to disallow client applications that use the connection factory from changing the configured ClientID through the `setClientID()` method of the `Connection` object.

It is recommended to always set the client identifier in deployed applications, either programmatically using the `setClientID()` method or using the `JMQConfiguredClientID` attribute of the `ConnectionFactory` object.

(If a ClientID value is not set, iMQ will provide a default value consisting of the client host's IP address. While this is done as a convenience to developers, it is not recommended for deployed applications.)

Table 4-3 Connection Factory Attributes: Client Identification

Attribute/property name	Description
`JMQDefaultUsername`	Specifies the default user name that will be used to authenticate with the broker. Default: `guest`
`JMQDefaultPassword`	Specifies the default password that will be used to authenticate with the broker. Default: `guest`
`JMQConfiguredClientID`	Specifies the value of an administratively configured ClientID. Default: `null`
`JMQDisableSetClientID`	Specifies if client application is prevented from changing the ClientID using the `setClientID()` method in the JMS API. Default: `false`

Reliability And Flow Control

These attributes determine the use and flow of iMQ control messages by the client runtime, especially broker acknowledgements (referred to as "Ack" in the attribute names).

Table 4-4 Connection Factory Attributes: Flow Reliability and Control

Attribute/property name	Description
`JMQAckOnProduce`	If set to `true`, broker acknowledges receipt of all JMS messages (persistent and non-persistent) from producing client, and producing client thread will block waiting for those acknowledgements (referred to as "Ack" in property name). If set to `false`, broker does not acknowledge receipt of any JMS message (persistent or non-persistent) from producing client, and producing client thread will not block waiting for broker acknowledgements. If not specified, broker acknowledges receipt of *persistent* messages only, and producing client thread will block waiting for those acknowledgements. Default: not specified
`JMQAckOnAcknowledge`	If set to `true`, broker acknowledges all consuming client acknowledgements, and consuming client thread will block waiting for such broker acknowledgements (referred to as "Ack" in property name). If set to `false`, broker does not acknowledge any consuming client acknowledgements, and consuming client thread will not block waiting for such broker acknowledgements. If not specified, broker acknowledges consuming client acknowledgements for `AUTO_ACKNOWLEDGE` and `CLIENT_ACKNOWLEDGE` mode (and consuming client thread will block waiting for such broker acknowledgements), but does not acknowledge consuming client acknowledgements for `DUPES_OK_ACKNOWLEDGE` mode (and consuming client thread will not block.) Default: not specified

Table 4-4 Connection Factory Attributes: Flow Reliability and Control *(Continued)*

Attribute/property name	Description
`JMQFlowControlCount`	Specifies the number of JMS messages in a metered batch. When this number of JMS messages is delivered to the client runtime, delivery is temporarily suspended, allowing any control messages that had been held up to be delivered. Payload message delivery is resumed upon notification by the client runtime, and continues until the count is again reached. If the count is set to 0 then there is no restriction in the number of JMS messages in a metered batch. A non-zero setting allows the client runtime to meter message flow so that iMQ control messages are not blocked by heavy JMS message delivery. Default: `100`
`JMQFlowControlIsLimited`	Specifies if `JMQFlowControlLimit` is enabled (only active if `JMQFlowControlCount` is a non-zero value). Default: `false`
`JMQFlowControlLimit`	Specifies a limit on the number of unconsumed messages that can be delivered to a client runtime (only used if `JMQFlowControlIsLimited=true`). When the number of JMS messages delivered to the client runtime (in accordance with the flow metering governed by `JMQFlowControlCount`) exceeds the limit, message delivery stops. It is resumed only when the number of unconsumed messages drops below the value set with this property. This limit prevents a consuming client that is taking a long time to process messages from being overwhelmed with pending messages that might cause it to run out of memory. Default: `1000`

Queue Browser Behavior

These attributes configure queue browsing for the client runtime.

Table 4-5 Connection Factory Attributes: Queue Browser Behavior

Attribute/property name	Description
`JMQQueueBrowserMax MessagesPerRetrieve`	Specifies the maximum number of messages that the client runtime will retrieve at one time, when browsing the contents of a queue destination. Default: `1000`
`JMQQueueBrowserRetrieve Timeout`	Specifies the maximum time that the client runtime will wait to retrieve messages, when browsing the contents of a queue destination, before throwing an exception. Default: `60000 milliseconds`.

Application Server Support

This attribute specifies the behavior of sessions running in an application server environment. For background see the "The Java Message Service (JMS) Specification" on page 19.

Table 4-6 Connection Factory Attributes: Application Server Support

Attribute/property name	Description
`JMQLoadMaxToServerSession`	Used only for JMS application server facilities. Specifies whether an iMQ ConnectionConsumer should load up to the `maxMessages` number of messages into a ServerSession's session (value=`true`), or load only a single message at a time (value=`false`). Default: `false`

JMS-defined Properties Support

These attributes specify which JMS-defined properties (see "The Java Message Service (JMS) Specification" on page 19) are supported by iMQ. JMS-defined properties are property names reserved by JMS, and which a JMS provider can choose to support. These properties enhance client application programming capabilities.

Table 4-7 Connection Factory Attributes: JMS-defined Properties Support

Attribute/property name	Description
`JMQSetJMSXUserID`	Specifies whether iMQ should set the JMS-defined property, `JMSXUserID` (identity of user sending the message), on produced messages. Default: `false`
`JMQSetJMSXAppID`	Specifies whether iMQ should set the JMS-defined property, `JMSXAppID` (identity of application sending the message), on produced messages. Default: `false`
`JMQSetJMSXProducerTXID`	Specifies whether iMQ should set the JMS-defined property, `JMSXProducerTXID` (transaction identifier of the transaction within which this message was produced), on produced messages. Default: `false`
`JMQSetJMSXConsumerTXID`	Specifies whether iMQ should set the JMS-defined property, `JMSXConsumerTXID` (transaction identifier of the transaction within which this message was consumed), on consumed messages. Default: `false`
`JMQSetJMSXRcvTimestamp`	Specifies whether iMQ should set the JMS-defined property, `JMSXRcvTimestamp` (the time the message is delivered to the consumer), on consumed messages. Default: `false`

Performance Factors

Because of the mechanisms by which messages are delivered to and from a broker, and because of the various iMQ control messages used to assure reliable delivery, there are a number of factors that affect message flow and consumption.

The factors that affect message throughput and performance are the following: delivery mode, message flow metering, message flow limits, acknowledgement mode, and number of sessions. These factors are quite distinct, yet interact in a way that can make it difficult to determine which of them might be impacting performance in any given client application.

Delivery mode The delivery mode specifies whether a message is to be delivered at most once (non-persistent) or once and only once (persistent). These different reliability requirements imply different degrees of overhead, especially for guaranteeing that persistent messages are not lost or delivered twice (as can happen in the case of a broker failure, a client runtime failure, or a lost connection). For example, the number of client and broker control messages flowing across a connection, and the handling of client and broker acknowledgements, have a strong impact on message throughput and performance.

Message flow metering Messages sent and received by client applications (JMS messages) and iMQ control messages pass over the same connection between client and broker. This can result in a bottleneck in the flow of control messages. For example, suppose a broker is flooding a connection with JMS messages being delivered to a client. That could create a delay in the delivery of control messages, such as broker acknowledgements. Hence, if you are experiencing delays, the cause might be that control messages are being held up by heavy delivery of JMS messages. To prevent this type of congestion, iMQ meters the flow of JMS messages across connections: JMS messages are batched so that only a set number can be delivered before delivery is temporarily suspended. Before JMS message delivery resumes, control messages that had previously been held up are delivered to the client. You can specify the number of messages allowed in such metered batches of JMS messages (see `JMQFlowControlCount` property, Table 4-4 on page 63). In cases of heavy JMS message delivery, decreasing the count should allow control messages to flow across a connection with less delay.

Message flow limits iMQ client runtime code can handle only a limited number of delivered JMS messages before encountering local resource limitations, such as memory. When this limit is approached, performance suffers. Hence, iMQ lets you limit the number of messages queued up in sessions awaiting consumption by controlling the flow of JMS messages to the client. You do this by specifying a threshold value (see `FlowControlLimit` property, Table 4-4 on page 63). If this threshold value is exceeded by delivery of a batch of JMS messages (see "Message flow metering" on page 67), the client runtime will wait until the number of un-consumed messages drops below this threshold before requesting that the delivery of JMS messages be resumed. Message delivery then continues until the threshold value is again exceeded.

Client acknowledgement mode The different client acknowledgement modes affect the number of client and broker acknowledgement messages passing over a connection:

- In the `AUTO_ACKNOWLEDGE` mode, a client acknowledgement and broker acknowledgement are required for each consumed message, and the session thread blocks waiting for the broker acknowledgement.
- In the `CLIENT_ACKNOWLEDGE` mode client acknowledgements and broker acknowledgements are batched (rather than being sent one by one). This conserves connection bandwidth and generally reduces the wait time for broker acknowledgements.
- In the `DUPS_OK_ACKNOWLEDGE` mode, throughput is improved even further, because client acknowledgements are batched and because the client thread does not block (broker acknowledgements are not requested). However, in this case, the same message can be delivered and consumed more than once.

Number of sessions and connections The number of messages queued up in a session, and the time it takes to process them, is a function of the number of message consumers in the session and the message load for each consumer. If a client application is exhibiting delays in producing or consuming messages, performance might be improved by redesigning the application to distribute message producers and consumers among a larger number of sessions or to distribute sessions among a larger number of connections.

Appendix A

Administered Object Attributes

This appendix provides reference tables for the attributes of the `ConnectionFactory` and `Destination` administered objects.

ConnectionFactory Administered Object

Table A-1 summarizes the configurable properties of a `ConnectionFactory` administered object. The attributes are presented in alphabetical order for quick reference. For groupings of these attributes in functional categories, and a description of each, see "iMQ Client Runtime Configurable Properties" on page 59.

Table A-1 Connection Factory Attributes

Attribute/property name	Type	Default Value	Reference
`JMQAckOnAcknowledge`	String	`not specified`	Table 4-5 on page 65
`JMQAckOnProduce`	String	`not specified`	Table 4-5 on page 65
`JMQAckTimeout`	String	`0 milliseconds`	Table 4-5 on page 65
`JMQBrokerHostName`	String	`localhost`	Table 4-1 on page 60
`JMQBrokerHostPort`	String	`7676`	Table 4-1 on page 60
`JMQConfiguredClientID`	String	`not specified`	Table 4-3 on page 62
`JMQConnectionType`	String	`TCP`	Table 4-1 on page 60
`JMQConnectionURL`	String	`http://localhost/servlet/HttpTunnelServlet`	Table 4-1 on page 60
`JMQDefaultPassword`	String	`guest`	Table 4-3 on page 62
`JMQDefaultUsername`	String	`guest`	Table 4-3 on page 62

Table A-1 Connection Factory Attributes *(Continued)*

Attribute/property name	Type	Default Value	Reference
`JMQDisableSetClientID`	String	`false`	Table 4-3 on page 62
`JMQFlowControlCount`	String	`100`	Table 4-4 on page 63
`JMQFlowControlIsLimited`	String	`false`	Table 4-4 on page 63
`JMQFlowControlLimit`	String	`1000`	Table 4-4 on page 63
`JMQLoadMaxToServerSession`	String	`false`	Table 4-7 on page 66
`JMQQueueBrowserMaxMessages PerRetrieve`	String	`1000`	Table 4-6 on page 65
`JMQQueueBrowserRetrieveTimeout`	String	`60,000 milliseconds`	Table 4-6 on page 65
`JMQReconnect`	String	`false`	Table 4-2 on page 61
`JMQReconnectDelay`	String	`30,000 milliseconds`	Table 4-2 on page 61
`JMQReconnectRetries`	String	`0`	Table 4-2 on page 61
`JMQSetJMSXAppID`	String	`false`	Table 4-7 on page 66
`JMQSetJMSXConsumerTXID`	String	`false`	Table 4-7 on page 66
`JMQSetJMSXProducerTXID`	String	`false`	Table 4-7 on page 66
`JMQSetJMSXRcvTimestamp`	String	`false`	Table 4-7 on page 66
`JMQSetJMSXUserID`	String	`false`	Table 4-7 on page 66
`JMQSSLIsHostTrusted`	String	`true`	Table 4-1 on page 60

For more information on using `ConnectionFactory` administered objects see Chapter 3, "Using Administered Objects."

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 only attribute is the physical destination's internal, provider-specific name. By creating a `Destination` object, you allow a client application's `MessageConsumer` and/or `MessageProducer` objects to access the corresponding physical destination.

Table A-2 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 "$".

For more information on `Destination` administered objects see Chapter 3, "Using Administered Objects."

Index

D

E

F

I

J

L

M

P

Q

S

T

U