Administrator's Guide

Sun™ ONE Message Queue

Version 3.5 Revised

817-3727-11
February 2004

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at http://www.sun.com/patents and one or more additional patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements. Use is subject to license terms. This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, Sun[tm] ONE, JDK, Java Naming and Directory Interface, Javadoc, JavaMail, JavaHelp, the Java Coffee Cup logo and the Sun[tm] ONE logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

This product is covered and controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuels relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plus des brevets américains listés à l'adresse http://www.sun.com/patents et un ou les brevets supplémentaires ou les applications de brevet en attente aux Etats - Unis et dans les autres pays.

L'utilisation est soumise aux termes de la Licence.

Cette distribution peut comprendre des composants développés par des tierces parties.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, Sun[tm] ONE, JDK, Java Naming and Directory Interface, Javadoc, JavaMail, JavaHelp, le logo Java Coffee Cup et le logo Sun[tm] ONE sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les mar ques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exlusivement par X/Open Company, Ltd.

Ce produit est soumis à la législation américaine en matière de contrôle des exportations et peut être soumis à la règlementation en vigueur dans d'autres pays dans le domaine des exportations et importations. Les utilisations, ou utilisateurs finaux, pour des armes nucléaires,des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers les pays sous embargo américain, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exhaustive, la liste de personnes qui fo nt objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

Contents

List of Figures **15**

List of Tables **17**

List of Procedures **21**

Preface **23**
Audience for This Guide 23
Organization of This Guide 24
Conventions 25
Text Conventions 25
Directory Variable Conventions 26
Other Documentation Resources 28
The MQ Documentation Set 28
Online Help 28
JavaDoc 29
Example Client Applications 29
The Java Message Service (JMS) Specification 29
Related Third-Party Web Site References 29

Chapter 1 Overview **31**
What Is Sun ONE Message Queue? 31
Product Editions 32
Platform Edition 33
Enterprise Edition 33
Enterprise Messaging Systems 34
Requirements of Enterprise Messaging Systems 34
Centralized vs. Peer to Peer Messaging 35
Messaging System Concepts 36
Message 36
Message Service Architecture 36
Message Delivery Models 37

The JMS Specification 37
JMS Message Structure 37
JMS Programming Model 38
JMS Administered Objects 39
JMS/J2EE Programming: Message-Driven Beans 40
Message-Driven Beans 41
J2EE Application Server Support 42
JMS Messaging Issues 42
JMS Provider Independence 43
Programming Domains 43
Client Identifiers 44
Reliable Messaging 45
Acknowledgements/Transactions 45
Persistent Storage 47
Performance Trade-offs 47
Message Selection 48
Message Order and Priority 48

Chapter 2 The MQ Messaging System 51
MQ Message Server 52
Broker 52
Connection Services 54
Message Router 58
Persistence Manager 63
Security Manager 66
Monitoring Service 70
Physical Destinations 76
Queue Destinations 76
Topic Destinations 78
Auto-Created (vs. Admin-Created) Destinations 78
Temporary Destinations 81
Multi-Broker Clusters (Enterprise Edition) 81
Multi-Broker Architecture 82
Using Clusters in Development Environments 85
Cluster Configuration Properties 85
MQ Client Runtime 86
Message Production 87
Message Consumption 87
MQ Administered Objects 88
Connection Factory Administered Objects 90
Destination Administered Objects 91
Overriding Attribute Values at Client Startup 92

Chapter 3 MQ Administration Tasks and Tools 93
MQ Administration Tasks 93
Development Environments 93
Production Environments 94
Setup Operations 94
To Set Up a Production Environment 94
Maintenance Operations 95
To Set Up a Production Environment 95
MQ Administration Tools 96
The Administration Console 96
Summary of Command Line Utilities 97
Command Line Syntax 98
Common Command Line Options 100

Chapter 4 Administration Console Tutorial 101
Getting Ready 102
Starting the Administration Console 102
To Start the Administration Console 103
Getting Help 104
To Display Administration Console Help Information 104
Working With Brokers 105
Starting a Broker 106
To Start a Broker 106
Adding a Broker 106
To Add a Broker to the Administration Console 107
Changing the Administrator Password 108
To Change the Administrator Password 108
Connecting to the Broker 108
To Connect to the Broker 108
Viewing Connection Services 110
To View Available Connection Services 110
Adding Physical Destinations to a Broker 111
To Add a Queue Destination to a Broker 112
Working With Physical Destinations 113
To View the Properties of a Physical Destination 113
To Purge Messages From a Destination 114
To Delete a Destination 114
Getting Information About Topic Destinations 115
Working with Object Stores 115
Adding an Object Store 116
To Add a File-system Object Store 116

Checking Object Store Properties . . . 119
To Display the Properties of an Object Store . . . 119
Connecting to an Object Store . . . 119
To Connect to an Object Store . . . 119
Adding a Connection Factory Administered Object . . . 119
To Add a Connection Factory to an Object Store . . . 120
Adding a Destination Administered Object . . . 121
To Add a Destination to an Object Store . . . 122
Administered Object Properties . . . 123
To View or Update the Properties of a Destination Object . . . 123
Updating Console Information . . . 124
Running the Sample Application . . . 124
To Run the HelloWorldMessageJNDI Application . . . 125

Chapter 5 Starting and Configuring a Broker . . . 127
Configuration Files . . . 127
Instance Configuration File . . . 128
Merging Property Values . . . 128
Property Naming Syntax . . . 129
Editing the Instance Configuration File . . . 129
Starting a Broker . . . 134
Syntax of the imqbrokerd Command . . . 135
Startup Examples . . . 136
To Start a Broker Instance That Uses the Default Broker Name and Configuration . . . 136
To Start a Broker Instance With a Trial Enterprise Edition License . . . 136
To Start a Named Broker Instance With Plugged-in Persistence . . . 136
Summary of imqbrokerd Options . . . 136
Working With Clusters (Enterprise Edition) . . . 140
Cluster Configuration Properties . . . 140
Connecting Brokers . . . 143
Connection Methods . . . 143
To Connect Brokers into a Cluster . . . 143
Secure Inter-Broker Connections . . . 143
To Configure Secure Connections Within a Cluster . . . 144
Managing Brokers in a Cluster . . . 144
Adding Brokers to a Cluster . . . 144
To Add a New Broker to an Existing Cluster . . . 144
Restarting a Broker in a Cluster . . . 145
To Restart a Broker That is Already a Member of an Existing Cluster . . . 145
Removing a Broker from a Cluster . . . 145
To Remove a Broker From an Existing Cluster . . . 145

Managing the Master Broker's Configuration Change Record 146
Backing up the Configuration Change Record 146
To Back Up the Configuration Change Record 146
Restoring the Configuration Change Record 146
To Restore the Master Broker in Case of Failure 146
Logging 147
Default Logging Configuration 147
Log Message Format 148
Changing the Logger Configuration 148
To Change the Logger Configuration for a Broker 148
Changing the Output Channel 149
Changing Log File Rollover Criteria 150

Chapter 6 Broker and Application Management 151
Command Utility 152
Syntax of the imqcmd Command 152
imqcmd Subcommands 152
Summary of imqcmd Options 154
Using imqcmd Commands 156
Example imqcmd Usage 157
Managing a Broker 157
Displaying Broker Information 159
Updating Broker Properties 160
Controlling the Broker's State 160
Pausing and Resuming a Broker 161
Shutting Down and Restarting a Broker 161
Displaying Broker Metrics 162
Managing Connection Services 162
Listing Connection Services 164
Displaying Connection Service Information 165
Updating Connection Service Properties 165
Displaying Connection Service Metrics 166
Pausing and Resuming a Connection Service 166
Getting Connection Information 167
Managing Destinations 168
Creating Destinations 170
Listing Destinations 173
Displaying Destination Information 173
Updating Destination Attributes 174
Displaying Destination Metrics 175
Pausing and Resuming Destinations 175
Purging Destinations 176
Destroying Destinations 176

Compacting Destinations 176
Monitoring a Destination's Disk Utilization 177
Reclaiming Unused Destination Disk Space 178
To Reclaim Unused Destination Disk Space 178
Managing Durable Subscriptions 179
Managing Transactions 180

Chapter 7 Managing Administered Objects 183
About Object Stores 184
LDAP Server Object Store 184
File-system Object Store 185
Administered Objects 186
Connection Factory Administered Object Attributes 187
Destination Administered Object Attributes 189
Object Manager Utility (imqobjmgr) 189
Syntax of the imqobjmgr Command 189
imqobjmgr Subcommands 190
Summary of imqobjmgr Command Options 190
Required Information 192
Using Command Files 193
Adding and Deleting Administered Objects 195
Adding a Connection Factory 195
Adding a Topic or Queue 196
Deleting Administered Objects 198
Getting Information 199
Listing Administered Objects 199
Information About a Single Object 200
Updating Administered Objects 200

Chapter 8 Managing Security 201
Authenticating Users 202
Using a Flat-File User Repository 202
Creating a User Repository 203
User Manager Utility (imqusermgr) 203
Groups 205
States 206
Format of User Names and Passwords 207
Populating and Managing a User Repository 207
Changing the Default Administrator Password 208
Using an LDAP Server for a User Repository 209
To Edit the Configuration File to use an LDAP Server 209

Authorizing Users: the Access Control Properties File 212
Creating an Access Control Properties File 212
Access Rules Syntax 213
Permission Computation 214
Connection Access Control 215
Destination Access Control 216
Destination Auto-Create Access Control 217
Encryption: Working With an SSL-based Service (Enterprise Edition) 218
Setting Up an SSL-based Service Over TCP/IP 218
To Set Up an SSL-based Connection Service 218
Step 1. Generating a Self-Signed Certificate 219
To Regenerate a Key Pair 220
Step 2. Enabling the SSL-based Service in the Broker 221
To Enable an SSL-based Service in the Broker 221
Step 3. Starting the Broker 221
Step 4. Configuring and Running SSL-based Clients 222
Setting Up an SSL Service Over HTTP 224
Using a Passfile 224

Chapter 9 Analyzing and Tuning an MQ Message Service 225
About Performance 225
The Performance Tuning Process 225
Aspects of Performance 226
Benchmarks 227
Baseline Use Patterns 228
Factors That Impact Performance 229
Application Design Factors that Impact Performance 230
Delivery Mode (Persistent/Non-persistent Messages) 232
Use of Transactions 233
Acknowledgement Mode 233
Durable vs. Non-Durable Subscriptions 234
Use of Selectors (Message Filtering) 236
Message Size 236
Message Body Type 237
Message Service Factors that Impact Performance 238
Hardware 239
Operating System 239
Java Virtual Machine (JVM) 239
Connections 239
Message Server Architecture 242
Broker Limits and Behaviors 242
Data Store Performance 243
Client Runtime Configuration 243

Monitoring an MQ Message Server 244
Monitoring Tools 244
MQ Command Utility (imqcmd) 244
To Use the metrics Subcommand 246
MQ Broker Log Files 249
To Use Log Files to Report Metrics Information 249
MQ Message-Based Monitoring API 250
To Set Up Message-based Monitoring 251
Choosing the Right Monitoring Tool 253
Description of Metrics Data 255
JVM Metrics 255
Broker-wide Metrics 256
Connection Service Metrics 258
Destination Metrics 260
Troubleshooting Performance Problems 262
Problem: Clients Can't Establish A Connection 263
Symptoms: 263
Possible Causes: 263
Problem: Connection Throughput is Too Slow 267
Symptoms: 267
Possible Causes: 267
Problem: Client Can't Create a Message Producer 269
Symptoms: 269
Possible Causes: 269
Problem: Message Production Is Delayed or Slowed 270
Symptoms: 270
Possible Causes: 270
Problem: Messages Backlogged in Message Server 273
Symptoms: 273
Possible Causes: 273
Problem: Message Server Throughput Is Sporadic 277
Symptoms: 277
Possible Causes: 277
Problem: Messages Not Reaching Consumers 279
Symptoms: 279
Possible Causes: 279

Adjusting Your Configuration To Improve Performance 280
System Adjustments 280
Solaris Tuning: CPU Utilization, Paging/Swapping/Disk I/O 280
Java Virtual Machine Adjustments 281
Tuning Transport Protocols 281
Tuning the File-based Persistent Store 285
Broker Adjustments 285
Memory Management: Increasing Broker Stability Under Load 285
Multiple Consumer Queue Performance 286
Client Runtime Message Flow Adjustments 286
Message Flow Metering 287
Message Flow Limits 287

Appendix A Location of MQ Data 291
Solaris 291
Linux 292
Windows 293

Appendix B Setting Up Plugged-in Persistence 295
Introduction 295
Plugging In a JDBC-accessible Data Store 296
To Plug in a JDBC-accessible Data Store 296
JDBC-related Broker Configuration Properties 297
Database Manager Utility (imqdbmgr) 301
Syntax of the imqdbmgr Command 302
imqdbmgr Subcommands 302
Summary of imqdbmgr Command Options 303

Appendix C HTTP/HTTPS Support (Enterprise Edition) 305
HTTP/HTTPS Support Architecture 305
Enabling HTTP Support 307
To Enable HTTP Support 307
Step 1. Deploying the HTTP Tunnel Servlet on a Web Server 307
Deploying as a Jar File 307
Deploying as a Web Archive File 308
Step 2. Configuring the httpjms Connection Service 308
To Activate the httpjms Connection Service 309

Step 3. Configuring an HTTP Connection 310
Configuring the Connection Factory 310
Using a Single Servlet to Access Multiple Brokers 311
Using an HTTP Proxy 311
Example 1: Deploying the HTTP Tunnel Servlet on Sun ONE Web Server 312
Deploying as a Jar File 312
To Add a Tunnel Servlet 312
To Configure a Virtual Path (Servlet URL) for a Tunnel Servlet 313
To Load the Tunnel Servlet at Web Server Startup 314
To Disable the Server Access Log 314
Deploying as a WAR File 314
To Deploy the http Tunnel Servlet as a WAR File 314
Example 2: Deploying the HTTP Tunnel Servlet on Sun ONE Application Server 7.0 315
Using the Deployment Tool 315
To Deploy the HTTP Tunnel Servlet in an Application Server 7.0 Environment 315
Modifying the server.policy File 316
To Modify the Application Server's server.policy File 317
Enabling HTTPS Support 317
To Enable HTTPS Support 317
Step 1. Generating a Self-signed Certificate for the HTTPS Tunnel Servlet 317
Step 2. Deploying the HTTPS Tunnel Servlet on a Web Server 318
Deploying as a Jar File 319
Deploying as a Web Archive File 319
Step 3. Configuring the httpsjms Connection Service 320
To Activate the httpsjms Connection Service 320
Step 4. Configuring an HTTPS Connection 321
Configuring JSSE 322
To Configure JSSE 322
Importing a Root Certificate 322
Configuring the Connection Factory 323
Using a Single Servlet to Access Multiple Brokers 323
Using an HTTP Proxy 324
Example 3: Deploying the HTTPS Tunnel Servlet on Sun ONE Web Server 324
Deploying as a Jar File 324
To Add a Tunnel Servlet 325
To Configure a Virtual Path (servlet URL) for a Tunnel Servlet 326
To Load the Tunnel Servlet at Web Server Startup 326
To Disable the Server Access Log 327
Deploying as a WAR File 327
To Modify the HTTPS Tunnel Servlet WAR File 327
To Deploy the https Tunnel Servlet as a WAR File 328

Example 4: Deploying the HTTPS Tunnel Servlet on Sun ONE Application Server 7.0 329
Using the Deployment Tool 329
To Deploy the HTTPS Tunnel Servlet in an Application Server 7.0 Environment 329
Modifying the server.policy file 330
To Modify the Application Server's server.policy File 330

Appendix D Using a Broker as a Windows Service 331
Running a Broker as a Windows Service 331
Service Administrator Utility (imqsvcadmin) 332
Syntax of the imqsvcadmin Command 332
imqsvcadmin Subcommands 332
Summary of imqsvcadmin Options 333
Removing the Broker Service 333
Reconfiguring the Broker Service 334
Using an Alternate Java Runtime 334
Querying the Broker Service 334
Troubleshooting 334
To See Logged Service Error Events 334

Appendix E The MQ Resource Adapter 335

Appendix F Technical Notes 337
System Clock Settings 337
Synchronization Recommended 337
Avoid Setting System Clocks Backwards 338
OS-Defined File Descriptor Limitations 338
Securing Persistent Data 339
Built-in Persistent Store 339
Plugged-in Persistent Store 339

Appendix G Stability of MQ Interfaces 341

Appendix H MQ Implementation of Optional JMS Functionality 345

Glossary 347

Index 351

List of Figures

Figure 1-1 Centralized vs. Peer to Peer Messaging 35
Figure 1-2 Message Service Architecture 36
Figure 1-3 JMS Programming Objects 38
Figure 1-4 Messaging with MDBs 41
Figure 2-1 MQ System Architecture 51
Figure 2-2 Broker Service Components 53
Figure 2-3 Connection Services Support 55
Figure 2-4 Persistence Manager Support 64
Figure 2-5 Security Manager Support 68
Figure 2-6 Monitoring Service Support 71
Figure 2-7 Multi-Broker (Cluster) Architecture 82
Figure 2-8 Messaging Operations 86
Figure 2-9 Message Delivery to MQ Client Runtime 88
Figure 3-1 Local and Remote Administration Utilities 97
Figure 5-1 Broker Configuration Files 129
Figure 9-1 Message Delivery Through an MQ System 229
Figure 9-2 Performance Impact of Delivery Modes 233
Figure 9-3 Performance Impact of Subscription Types 235
Figure 9-4 Performance Impact of a Message Size 237
Figure 9-5 Transport Protocol Speeds 240
Figure 9-6 Performance Impact of Transport Protocol 241
Figure 9-7 Effect of Changing `inbufsz` on a 1k (1024 bytes) Packet 283
Figure 9-8 Effect of Changing `outbufsz` on a 1k (1024 bytes) Packet 284
Figure C-1 HTTP/HTTPS Support Architecture 306

List of Tables

Table 1 Book Contents 24
Table 2 Document Conventions 25
Table 3 MQ Directory Variables 26
Table 4 MQ Documentation Set 28
Table 1-1 JMS Programming Objects 44
Table 2-1 Main Broker Service Components and Functions 53
Table 2-2 Connection Services Supported by a Broker 54
Table 2-3 Connection Service Properties 57
Table 2-4 Message Router Properties 62
Table 2-5 Persistence Manager Properties 66
Table 2-6 Security Manager Properties 69
Table 2-7 Logging Categories 72
Table 2-8 Metrics Topic Destinations 73
Table 2-9 Monitoring Service Properties 74
Table 2-10 Auto-create Configuration Properties 79
Table 2-11 Destination Attributes 91
Table 3-1 Common MQ Command Line Options 100
Table 5-1 Broker Instance Configuration Properties 130
Table 5-2 `imqbrokerd` Options 136
Table 5-3 Cluster Configuration Properties 141
Table 5-4 `imqbrokerd` Logger Options and Corresponding Properties 149
Table 6-1 `imqcmd` Subcommands 152
Table 6-2 `imqcmd` Options 154
Table 6-3 `imqcmd` Subcommands Used to Manage a Broker 158
Table 6-4 Broker Properties Updated by `imqcmd` 160
Table 6-5 `imqcmd` Subcommands Used to Manage Connection Services 163
Table 6-6 Connection Services Supported by a Broker 164

Table 6-7 Connection Service Properties Updated by `imqcmd` . . . 165
Table 6-8 `imqcmd` Subcommands Used to Manage Connection Services . . . 167
Table 6-9 `imqcmd` Subcommands Used to Manage Destinations . . . 168
Table 6-10 Destination Attributes . . . 171
Table 6-11 Destination disk Utilization Metrics . . . 177
Table 6-12 `imqcmd` Subcommands Used to Manage Durable Subscriptions . . . 179
Table 6-13 `imqcmd` Subcommands Used to Manage Transactions . . . 180
Table 7-1 LDAP Object Store Attributes . . . 184
Table 7-2 File-system Object Store Attributes . . . 186
Table 7-3 Connection Factory Administered Object Attributes . . . 187
Table 7-4 Destination Administered Object Attributes . . . 189
Table 7-5 `imqobjmgr` Subcommands . . . 190
Table 7-6 `imqobjmgr` Options . . . 190
Table 7-7 Naming Convention Examples . . . 196
Table 8-1 Initial Entries in User Repository . . . 203
Table 8-2 `imqusermgr` Subcommands . . . 204
Table 8-3 `imqusermgr` Options . . . 205
Table 8-4 Invalid Characters for User Names and Passwords . . . 207
Table 8-5 LDAP-related Properties . . . 210
Table 8-6 Syntactic Elements of Access Rules . . . 213
Table 8-7 Elements of Destination Access Control Rules . . . 216
Table 8-8 Keystore Properties . . . 220
Table 8-9 Passwords in a Passfile . . . 224
Table 9-1 Comparison of High Reliability and High Performance Scenarios . . . 231
Table 9-2 `imqcmd metrics` Subcommand Syntax . . . 245
Table 9-3 `imqcmd metrics` Subcommand Options . . . 245
Table 9-4 `imqcmd query` Subcommand Syntax . . . 248
Table 9-5 Metrics Topic Destinations . . . 251
Table 9-6 Pros and Cons of Metrics Monitoring Tools . . . 254
Table 9-7 JVM Metrics . . . 255
Table 9-8 Broker-wide Metrics . . . 256
Table 9-9 Connection Service Metrics . . . 258
Table 9-10 Destination Metrics . . . 260
Table A-1 Location of MQ Data on Solaris . . . 291
Table A-2 Location of MQ Data on Linux . . . 292
Table A-3 Location of MQ Data on Windows . . . 293
Table B-1 JDBC-related Properties . . . 298

Table B-2 imqdbmgr Subcommands 302
Table B-3 imqdbmgr Options 303
Table C-1 httpjms Connection Service Properties 309
Table C-2 Servlet Arguments for Deploying HTTP Tunnel Servlet Jar File 313
Table C-3 httpsjms Connection Service Properties 321
Table C-4 Servlet Arguments for Deploying HTTPS Tunnel Servlet Jar File 325
Table D-1 imqsvcadmin Subcommands 332
Table D-2 imqsvcadmin Options 333
Table G-1 Stability of MQ Interfaces 341
Table G-2 Interface Stability Classification Scheme 343
Table H-1 Optional JMS Functionality 345

List of Procedures

To Set Up a Production Environment 94
To Set Up a Production Environment 95
To Start the Administration Console 103
To Display Administration Console Help Information 104
To Start a Broker 106
To Add a Broker to the Administration Console 107
To Change the Administrator Password 108
To Connect to the Broker 108
To View Available Connection Services 110
To Add a Queue Destination to a Broker 112
To View the Properties of a Physical Destination 113
To Purge Messages From a Destination 114
To Delete a Destination 114
To Add a File-system Object Store 116
To Display the Properties of an Object Store 119
To Connect to an Object Store 119
To Add a Connection Factory to an Object Store 120
To Add a Destination to an Object Store 122
To View or Update the Properties of a Destination Object 123
To Run the HelloWorldMessageJNDI Application 125
To Start a Broker Instance That Uses the Default Broker Name and Configuration 136
To Start a Broker Instance With a Trial Enterprise Edition License 136
To Start a Named Broker Instance With Plugged-in Persistence 136
To Connect Brokers into a Cluster 143
To Configure Secure Connections Within a Cluster 144
To Add a New Broker to an Existing Cluster 144
To Restart a Broker That is Already a Member of an Existing Cluster 145

To Remove a Broker From an Existing Cluster 145
To Back Up the Configuration Change Record 146
To Restore the Master Broker in Case of Failure 146
To Change the Logger Configuration for a Broker 148
To Reclaim Unused Destination Disk Space 178
To Edit the Configuration File to use an LDAP Server 209
To Set Up an SSL-based Connection Service 218
To Regenerate a Key Pair 220
To Enable an SSL-based Service in the Broker 221
To Use the metrics Subcommand 246
To Use Log Files to Report Metrics Information 249
To Set Up Message-based Monitoring 251
To Plug in a JDBC-accessible Data Store 296
To Enable HTTP Support 307
To Activate the httpjms Connection Service 309
To Add a Tunnel Servlet 312
To Configure a Virtual Path (Servlet URL) for a Tunnel Servlet 313
To Load the Tunnel Servlet at Web Server Startup 314
To Disable the Server Access Log 314
To Deploy the http Tunnel Servlet as a WAR File 314
To Deploy the HTTP Tunnel Servlet in an Application Server 7.0 Environment 315
To Modify the Application Server's server.policy File 317
To Enable HTTPS Support 317
To Activate the httpsjms Connection Service 320
To Configure JSSE 322
To Add a Tunnel Servlet 325
To Configure a Virtual Path (servlet URL) for a Tunnel Servlet 326
To Load the Tunnel Servlet at Web Server Startup 326
To Disable the Server Access Log 327
To Modify the HTTPS Tunnel Servlet WAR File 327
To Deploy the https Tunnel Servlet as a WAR File 328
To Deploy the HTTPS Tunnel Servlet in an Application Server 7.0 Environment 329
To Modify the Application Server's server.policy File 330
To See Logged Service Error Events 334

Preface

This book, the Sun™ Open Net Environment (Sun ONE) Message Queue (MQ) 3.5 *Administrator's Guide*, provides the background and information needed to perform administration tasks for an MQ messaging system.

This preface contains the following sections:

- "Audience for This Guide" on page 23
- "Organization of This Guide" on page 24
- "Conventions" on page 25
- "Other Documentation Resources" on page 28

Audience for This Guide

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

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

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

Organization of This Guide

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

Table 1 Book Contents

Chapter	Description
Chapter 1, “Overview”	Presents a high-level conceptual overview of MQ messaging systems and terminology.
Chapter 2, “The MQ Messaging System”	Describes the MQ messaging system, with special emphasis on the MQ broker and the MQ client runtime that together provide messaging services.
Chapter 3, “MQ Administration Tasks and Tools”	Describes MQ administration tasks and tools, and introduces the command line utilities used for administration, and their common features.
Chapter 4, “Administration Console Tutorial”	Provides a hands-on tutorial to acquaint you with the Administration Console, a graphical interface to the MQ message server.
Chapter 5, “Starting and Configuring a Broker”	Explains how to start up and configure an MQ broker and a broker cluster.
Chapter 6, “Broker and Application Management”	Explains how to perform (application-independent) tasks related to managing MQ brokers, as well as tasks used to manage messaging applications.
Chapter 7, “Managing Administered Objects”	Explains how to perform tasks related to creating and managing MQ administered objects.
Chapter 8, “Managing Security”	Explains how to perform security tasks related to applications, such as managing authentication, authorization, and encryption.
Chapter 9, “Analyzing and Tuning an MQ Message Service”	Describes techniques for monitoring and analyzing message server performance and explains how to tune the message server to optimize its performance.
Appendix A, “Location of MQ Data”	Describes the location of various categories of MQ data.
Appendix B, “Setting Up Plugged-in Persistence”	Explains how to set up MQ to use JDBC-compliant database to perform persistence functions.
Appendix C, “HTTP/HTTPS Support (Enterprise Edition)”	Explains how to set up HTTP connection services between a messaging client and the MQ message server.
Appendix D, “Using a Broker as a Windows Service”	Explains how to use the MQ Service Administration utility (`imqsvcadmin`) to install, query, and remove the broker (running as an Windows service).

Table 1 Book Contents *(Continued)*

Chapter	Description
Appendix F, “Technical Notes”	Provides a number of specialized technical notes relevant to topics in this book, but not part of MQ-specific administration.
Appendix E, “The MQ Resource Adapter”	Describes what the MQ resource adapter is, how to deploy it, and how to configure and use it.
Appendix H, “MQ Implementation of Optional JMS Functionality”	Describes how the MQ product handles each of the items listed in the JMS specification as optional for a JMS provider to implement. .
Appendix G, “Stability of MQ Interfaces”	Describes the stability of various MQ interfaces.
“Glossary”	Defines terms used in MQ documentation.

Conventions

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

Text Conventions

Table 2 Document Conventions

Format	Description
italics	Italicized text represents a placeholder. Substitute an appropriate clause or value where you see italic text. Italicized text is also used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and 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 (IMQ_HOME), or acronyms (MQ, JSP).
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.

Table 2 Document Conventions *(Continued)*

Format	Description
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Directory Variable Conventions

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

Table 3 MQ Directory Variables

Variable	Description
IMQ_HOME	This is generally used in MQ documentation to refer to the MQ base directory (root installation directory): • On Solaris, there is no root MQ installation directory. Therefore, `IMQ_HOME` is not used in MQ documentation to refer to file locations on Solaris. • On Solaris, for Sun ONE Application Server, Evaluation Edition, the root MQ installation directory is: *Application Server base directory*`/imq`. • On Windows, the root MQ installation directory is set by the MQ installer (by default, as `C:\Program Files\Sun\MessageQueue3`). • On Windows, for Sun ONE Application Server, the root MQ installation directory is: *Application Server base directory*`/imq`. • On Linux, there is no root MQ installation directory. Therefore, `IMQ_HOME` is not used in MQ documentation to refer to file locations on Linux.

Table 3 MQ Directory Variables *(Continued)*

Variable	Description
IMQ_VARHOME	This is the /var directory in which MQ temporary or dynamically-created configuration and data files are stored. It can be set as an environment variable to point to any directory. • On Solaris, IMQ_VARHOME defaults to the /var/imq directory. • On Solaris, for Sun ONE Application Server, Evaluation Edition, IMQ_VARHOME defaults to the IMQ_HOME/var directory. • On Windows IMQ_VARHOME defaults to the IMQ_HOME\var directory. • On Windows, for Sun ONE Application Server, IMQ_VARHOME defaults to the IMQ_HOME\var directory. • On Linux, IMQ_VARHOME defaults to the /var/opt/imq directory
IMQ_JAVAHOME	This is an environment variable that points to the location of the Java™ runtime (JRE) required by MQ executables: • On Solaris, IMQ_JAVAHOME defaults to the /usr/j2se/jre directory, but a user can optionally set the value to wherever the required JRE resides. • On Windows, IMQ_JAVAHOME defaults to IMQ_HOME\jre, but a user can optionally set the value to wherever the required JRE resides. • On Linux, MQ first looks for the java runtime in the /usr/java/j2sdk*Version* directory, and then looks in the /usr/java/j2re*Version* directory, but a user can optionally set the value of IMQ_JAVAHOME to wherever the required JRE resides.

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

Other Documentation Resources

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

The MQ Documentation Set

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

Table 4 MQ Documentation Set

Document	Audience	Description
MQ *Installation Guide*	Developers and administrators	Explains how to install MQ software on Solaris, Linux, and Windows platforms.
MQ *Release Notes*	Developers and administrators	Includes descriptions of new features, limitations, and known bugs, as well as technical notes.
MQ *Administrator's Guide*	Administrators, also recommended for developers	Provides background and information needed to perform administration tasks using MQ administration tools.
MQ *Java Client Developer's Guide*	Developers	Provides a quick-start tutorial and programming information for developers of Java client programs using the MQ implementation of the JMS and SOAP/JAXM specifications.
MQ *C Client Developer's Guide*	Developers	Provides programming and reference documentation for developers of C client programs using the C interface (C-API) to the .MQ message service.

Online Help

MQ includes command line utilities for performing MQ message service administration tasks. To access the online help for these utilities, see "Common Command Line Options" on page 100.

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

JavaDoc

MQ Java client API (including the JMS API) documentation in JavaDoc format, is provided in a directory that depends upon the operating system (see Appendix A, "Location of MQ Data").

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

Example Client Applications

A number of example applications that provide sample client application code are included in a directory that depends upon the operating system (see Appendix A, "Location of MQ Data").

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

The Java Message Service (JMS) Specification

The JMS specification can be found at the following location:

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

The specification includes sample client code.

Related Third-Party Web Site References

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

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

Chapter 1

Overview

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

What Is Sun ONE Message Queue?

The MQ product is a standards-based solution for reliable, asynchronous messaging for distributed applications. MQ is an enterprise messaging system that implements the Java™ Message Service (JMS) open standard: in fact it serves as the JMS Reference Implementation. However MQ is also a full-featured JMS provider with enterprise-strength features.

The JMS specification describes a set of messaging semantics and behaviors, and an application programming interface (API), that provide a common way for Java language applications to create, send, receive, and read messages in a distributed environment (see "JMS Programming Model" on page 38). In addition to supporting Java messaging applications, MQ also provides a C language interface to the MQ message service (the MQ C-API).

With Sun ONE Message Queue software, processes running on different platforms and operating systems can connect to a common MQ message service (see "Message Service Architecture" on page 36) 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 reliably communicate across a network.

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

Centralized administration. Provides both command-line and GUI tools for administering an MQ message service and managing application-dependent entities, such as destinations, transactions, durable subscriptions, and security. MQ also supports remote monitoring of the MQ message service.

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

Client connection failover. Automatically restores a failed client connection to a MQ message server.

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

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

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

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

See *Appendix H, "MQ Implementation of Optional JMS Functionality"* for documentation of JMS compliance-related issues.

Product Editions

Sun ONE Message Queue is available in two editions: Platform and Enterprise—each corresponding to a different feature set and licensed capacity, as described below. (Instructions for upgrading MQ from one edition to another are in the MQ *Installation Guide*.)

Platform Edition

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

- **a basic license.** This license provides basic JMS support (it's a full JMS provider), but does *not* include such enterprise features as load balancing (multi-broker message service), HTTP/HTTPS connections, secure connection services, scalable connection capability, client connection failover, queue delivery to multiple consumers, remote message-based monitoring, and C-API support. The license has an unlimited duration, and can therefore be used in less demanding production environments.
- **a 90-day trial enterprise license.** This license includes all enterprise features (such as support for multi-broker message services, HTTP/HTTPS connections, secure connection services, scalable connection capability, client connection failover, queue delivery to multiple consumers, remote message-based monitoring, and C-API support) not included in the basic license. However, the license has a limited 90-day duration enforced by the software, making it suitable for evaluating the enterprise features available in the Enterprise Edition of the product (see "Enterprise Edition").

NOTE The 90-day trial license can be enabled by starting the MQ message server—an MQ broker instance—as described in "To Start a Broker Instance With a Trial Enterprise Edition License" on page 136.

Enterprise Edition

This edition is for deploying and running messaging applications in a production environment. It includes support for multi-broker message services, HTTP/HTTPS connections, secure connection services, scalable connection capability, client connection failover, queue delivery to multiple consumers, remote message-based monitoring, and C-API support. You can also use the Enterprise Edition for developing, debugging, and load testing messaging applications and components. The Enterprise Edition has an unlimited duration license that places no limit on the number of brokers in a multi-broker message service, but is based on the number of CPUs that are used.

Enterprise Messaging Systems

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

Requirements of Enterprise Messaging Systems

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

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

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

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

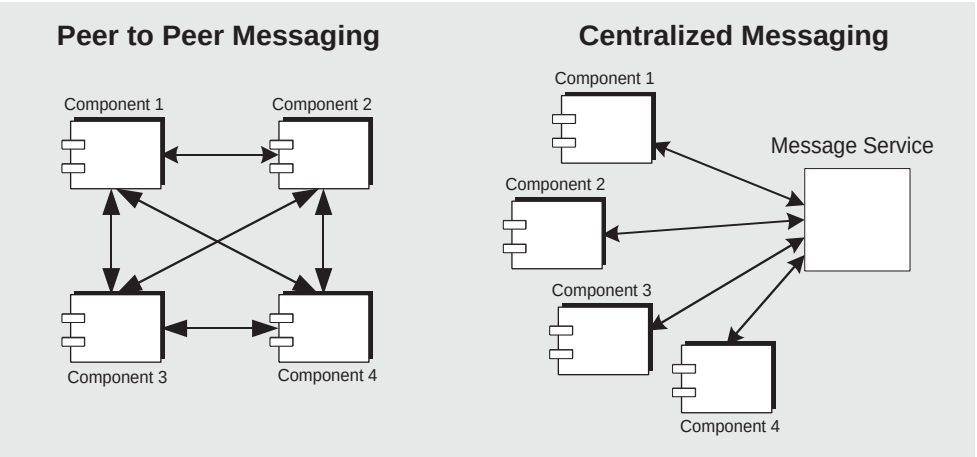
Scalability. The messaging system must be able to accommodate increasing loads—increasing numbers of users and increasing numbers of messages—without a substantial loss of performance or message throughput. As businesses and applications expand, this becomes a very important requirement.

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

Centralized vs. Peer to Peer Messaging

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

Figure 1-1 Centralized vs. Peer to Peer Messaging



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

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

Messaging System Concepts

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

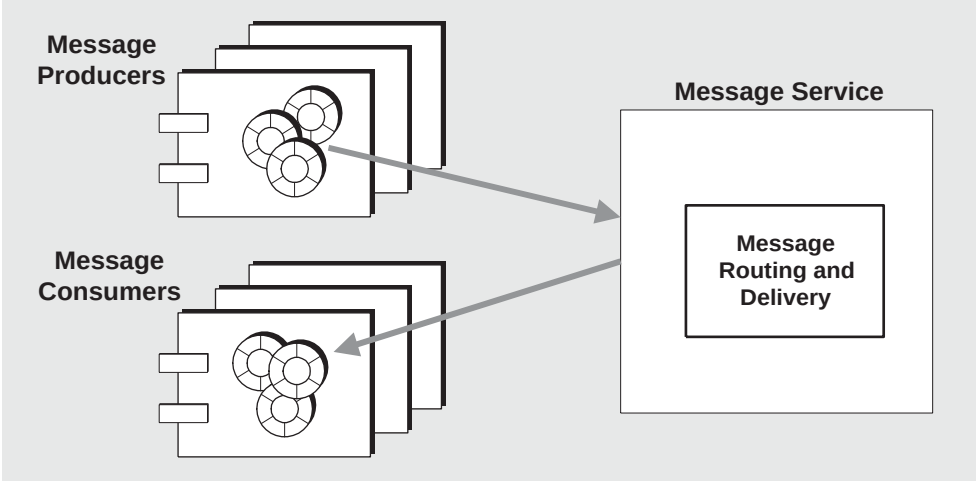
Message

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

Message Service Architecture

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

Figure 1-2 Message Service Architecture



Message Delivery Models

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

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

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

Historically, messaging systems supported various combinations of these two message delivery models. The Java Message Service (JMS) specification creates standard semantics for messaging with an API for Java programming. It supports both the point-to-point and publish/subscribe message delivery models (see "Programming Domains" on page 43).

The JMS Specification

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

JMS Message Structure

A JMS message is composed of three parts: a header, properties, and a body.

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

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

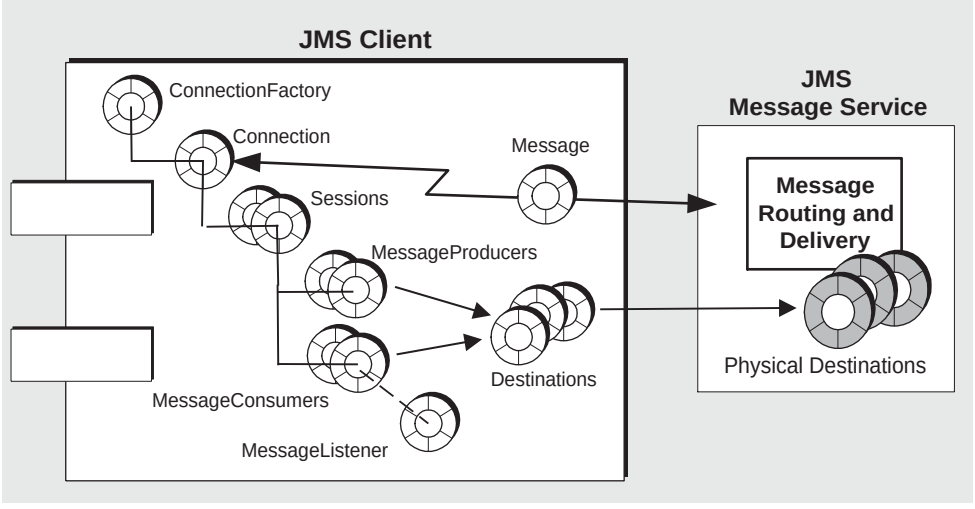
Message body The message body contains the actual data to be exchanged. JMS supports six body types.

JMS Programming Model

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

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

Figure 1-3 JMS Programming Objects



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

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

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

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

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

JMS Administered Objects

The JMS specification facilitates provider-independent clients by specifying *administered objects* that encapsulate provider-specific configuration information.

Two of the objects described in the "JMS Programming Model" on page 38 depend on how a JMS provider implements a JMS message service. The connection factory object depends on the underlying protocols and mechanisms used by the provider to deliver messages, and the destination object depends on the specific naming conventions and capabilities of the physical destinations used by the provider.

Normally these provider-specific characteristics would make JMS client code dependent on a specific JMS implementation. However, the JMS specification requires that provider-specific implementation and configuration information be encapsulated in connection factory and destination objects that can then be accessed in a standardized, non-provider-specific way.

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

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

JMS/J2EE Programming: Message-Driven Beans

In addition to the general JMS client programming model introduced in "JMS Programming Model" on page 38, there is a more specialized adaptation of JMS used in the context of Java 2 Platform, Enterprise Edition (J2EE platform) applications. This specialized JMS client is called a *message-driven bean* and is one of a family of Enterprise JavaBeans (EJB) components specified in the EJB 2.0 Specification (`http://java.sun.com/products/ejb/docs.html`).

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

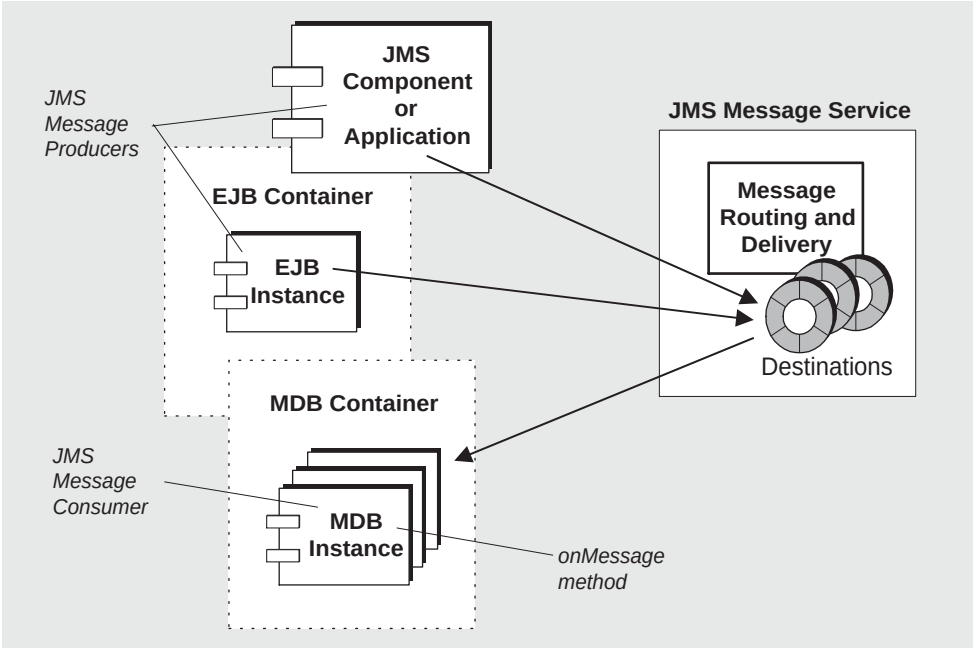
However, asynchronous messaging is a requirement of many enterprise applications. Most such applications require that server-side components be able to communicate and respond to each other without tying up server resources. Hence, the need for an EJB component that can receive messages and consume them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

Message-Driven Beans

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

Message-driven Bean The MDB is a JMS message consumer that implements the JMS `MessageListener` interface. The `onMessage` method (written by the MDB developer) is invoked when a message is received by the MDB container. The `onMessage()` method consumes the message, just as the `onMessage()` method of a standard `MessageListener` object would. You do not remotely invoke methods on MDBs—like you do on other EJB components—therefore there are no home or remote interfaces associated with them. The MDB can consume messages from a single destination. The messages can be produced by standalone JMS applications, JMS components, EJB components, or Web components, as shown in Figure 1-4.

Figure 1-4 Messaging with MDBs



MDB Container The MDB is supported by a specialized EJB container, responsible for creating instances of the MDB and setting them up for asynchronous consumption of messages. This involves setting up a connection with the message service (including authentication), creating a pool of sessions

associated with a given destination, and managing the distribution of messages as they are received among the pool of sessions and associated MDB instances. Since the container controls the life-cycle of MDB instances, it manages the pool of MDB instances so as to accommodate incoming message loads.

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

J2EE Application Server Support

In J2EE architecture (see the J2EE Platform Specification located at `http://java.sun.com/j2ee/download.html#platformspec`), EJB containers are hosted by J2EE application servers. An application server provides resources needed by the various containers: transaction managers, persistence managers, name services, and, in the case of messaging and MDBs, a JMS provider.

In the Sun ONE Application Server, JMS messaging resources are provided by Sun ONE Message Queue:

- For Sun ONE Application Server 7.0, an MQ messaging system is integrated into the application server as its native JMS provider.
- For the Sun J2EE 1.4 Application Server, MQ is plugged into the application server as an embedded JMS resource adapter (see Appendix E, "The MQ Resource Adapter").
- For future releases of the Application Server, MQ will be plugged into the application server using standard resource adapter deployment and configuration methods.

JMS Messaging Issues

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

JMS Provider Independence

JMS specifies the use of administered objects (see "JMS Administered Objects" on page 39) to support the development of client applications that are portable to other JMS providers. Administered objects allow JMS clients to use logical names to look up and reference provider-specific objects. In this way client code does not need to know specific naming or addressing syntax or configurable properties used by a provider. This makes the code provider-independent.

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

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

For more information on administered objects, see "MQ Administered Objects" on page 88.

Programming Domains

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

point-to-point (Queue Destinations) A message is delivered from a producer to one consumer. In this delivery model, the destination is a *queue*. Messages are first delivered to the queue destination, then delivered from the queue, one at a time, depending on the queue's delivery policy (see "Queue Destinations" on page 76), to one of the consumers registered for the queue. Any number of producers can send messages to a queue destination, but each message is guaranteed to be delivered to—and successfully consumed by—only *one* consumer. If there are no consumers registered for a queue destination, the queue holds messages it receives, and delivers them when a consumer registers for the queue.

Publish/Subscribe (Topic destinations) A message is delivered from a producer to any number of consumers. In this delivery model, the destination is a *topic*. Messages are first delivered to the topic destination, then delivered to *all* active consumers that have *subscribed* to the topic. Any number of producers can send messages to a topic destination, and each message can be delivered to any number of subscribed consumers. Topic destinations also support the notion of *durable subscriptions.* A durable subscription represents a consumer that is registered with

the topic destination but can be inactive at the time that messages are delivered. When the consumer subsequently becomes active, it receives the messages. If there are no consumers registered for a topic destination, the topic does not hold messages it receives, unless it has inactive consumers with durable subscriptions.

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

Table 1-1 JMS Programming Objects

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

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

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

Client Identifiers

JMS providers must support the notion of a *client identifier*, which associates a JMS client's connection to a message service with state information maintained by the message service on behalf of the client. By definition, a client identifier is unique, and applies to only one user at a time. Client identifiers are used in combination with a durable subscription name (see "Publish/Subscribe (Topic destinations)" on page 43) to make sure that each durable subscription corresponds to only one user.

The JMS specification allows client identifiers to be set by the client through an API method call, but recommends setting it administratively using a connection factory administered object (see "JMS Administered Objects" on page 39). If hard wired into a connection factory, however, each user would then need an individual connection factory to have a unique identity.

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

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

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

Reliable Messaging

JMS defines two *delivery modes*:

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

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

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

Acknowledgements/Transactions

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

Acknowledgements

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

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

Local Transactions

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

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

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

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

Distributed Transactions

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

Support for distributed transactions means that messaging clients can participate in distributed transactions through the XAResource interface defined by JTA. This interface defines a number of methods for implementing two-phase commit. While the API calls are made on the client side, the MQ broker tracks the various send and receive operations within the distributed transaction, tracks the transactional state, and completes the messaging operations only in coordination with a distributed transaction manager—provided by a Java Transaction Service (JTS).

As with local transactions, the client can handle exceptions by ignoring them, retrying operations, or rolling back an entire distributed transaction.

MQ implements support for distributed transactions through an XA connection factory, which lets you create XA connections, which in turn lets you create XA sessions (see "JMS Programming Model" on page 38). In addition, support for distributed transactions requires either a third party JTS or a J2EE-compliant Application Server (that provides JTS).

Persistent Storage

The other important aspect of reliability is assuring that once persistent messages are delivered to their destinations, a message service does not lose them before they are delivered to consumers. This means that upon delivery of a persistent message to its destination, the message service must place it in a persistent data store (see "Persistence Manager" on page 63). If the message service goes down for any reason, it can recover the message and deliver it to the appropriate consumers. While this adds overhead to message delivery, it also adds reliability.

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

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

Performance Trade-offs

The more reliable the delivery of messages, the more overhead and bandwidth are required to achieve it. The trade-off between reliability and performance is a significant design consideration. You can maximize *performance* by choosing to produce and consume non-persistent messages. On the other hand, you can

maximize *reliability* by producing and consuming persistent messages and using transacted sessions. Between these extremes are a number of options, depending on the needs of an application, including the use of MQ-specific connection and acknowledgement properties (see the MQ *Java Client Developer's Guide*). These trade-offs are discussed more fully in "Application Design Factors that Impact Performance" on page 230.

Message Selection

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

Message Order and Priority

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

Beyond this, the ordering of messages consumed by a client application can have only a rough relationship to the order in which they were produced. This is because the delivery of messages to destinations and the delivery from those destinations can depend on a number of issues that affect timing, such as the order in which the messages are sent, the sessions (connections) from which they are sent, whether the messages are persistent, the lifetime of the messages, the priority of the messages, the message delivery policy of queue destinations (see "Queue Destinations" on page 76), and message service availability.

In the case of an MQ message server using multiple interconnected brokers (see "Multi-Broker Clusters (Enterprise Edition)" on page 81) the ordering of messages consumed by a client is further complicated by the fact that the order of delivery from destinations on different brokers is indeterminate. Hence, a message delivered by one broker might precede a message delivered by another broker even though the latter might have received the message first.

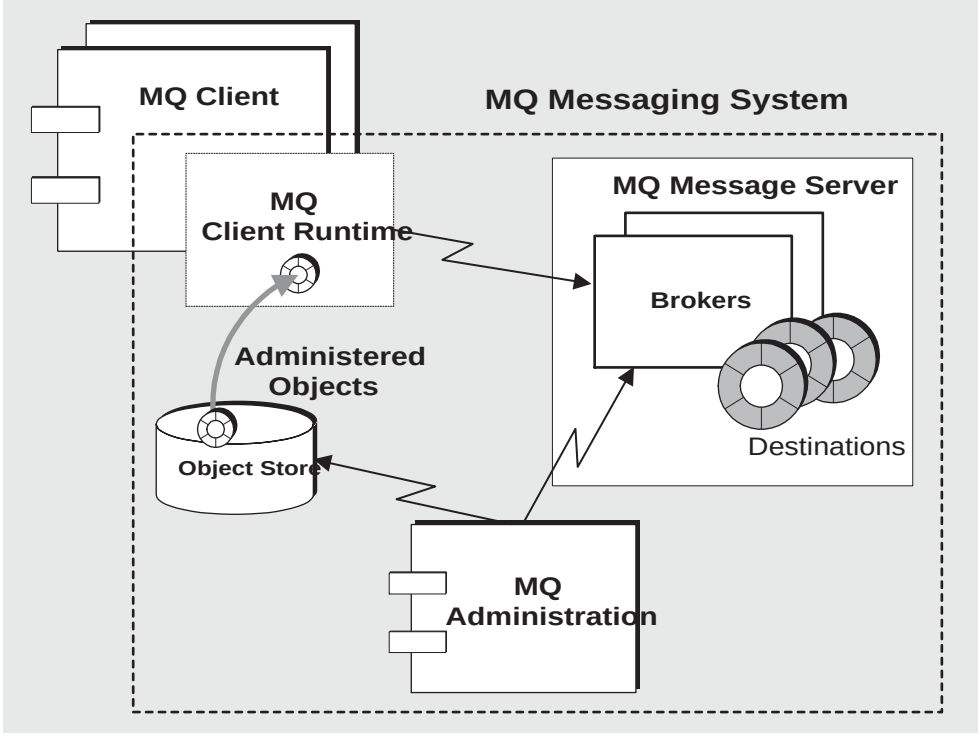
In any case, for a given consumer, precedence is given for higher priority messages over lower priority messages.

Chapter 2

The MQ Messaging System

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

Figure 2-1 MQ System Architecture



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

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

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

MQ Message Server

This section describes the different parts of the MQ message server shown in Figure 2-1 on page 51. These include the following:

Broker An MQ broker provides delivery services for an MQ messaging system. Message delivery relies upon a number of supporting components that handle connection services, message routing and delivery, persistence, security, and logging (see "Broker" for more information). A message server can employ one or more broker instances (see "Multi-Broker Clusters (Enterprise Edition)" on page 81).

Physical Destination Delivery of a message is a two-phase process—delivery from a producing client to a physical destination maintained by a broker, followed by delivery from the destination to one or more consuming clients. Physical destinations represent locations in a broker's physical memory and/or persistent storage (see "Physical Destinations" on page 76 for more information).

Broker

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

To perform this complex set of functions, a broker uses a number of different internal components, each with a specific role in the delivery process. These broker components are illustrated in Figure 2-2 and described briefly in Table 2-1. The Message Router component performs the key message routing and delivery service, and the others provide important support services upon which the Message Router depends.

Figure 2-2 Broker Service Components

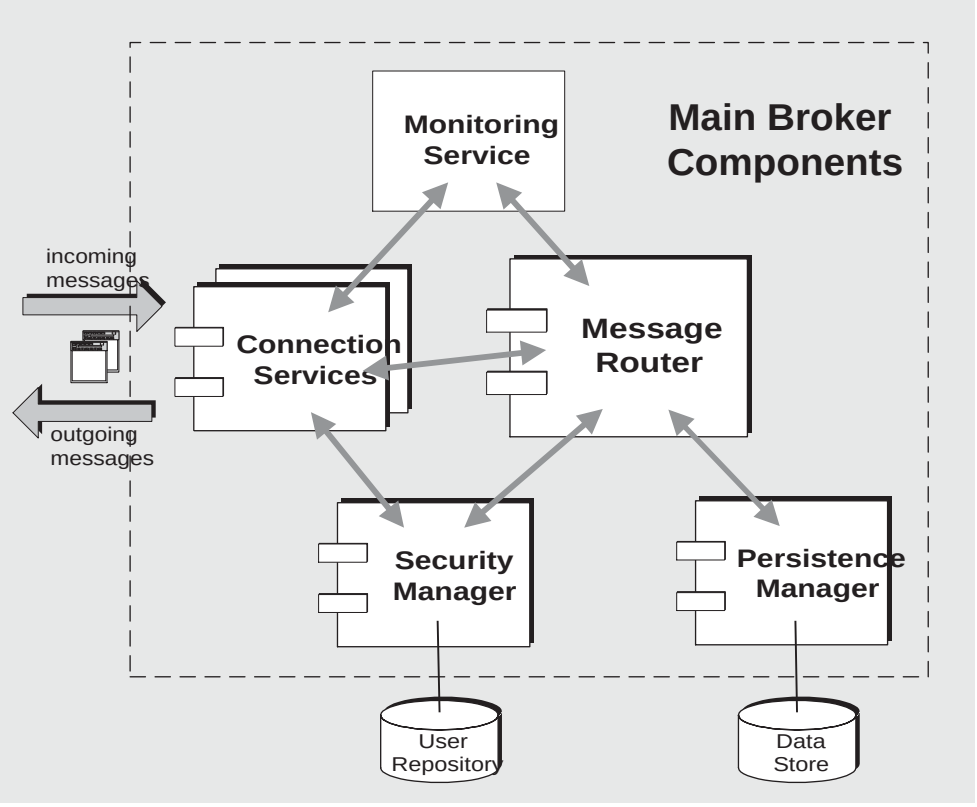


Table 2-1 Main Broker Service Components and Functions

Component	Description/Function
Message Router	Manages the routing and delivery of messages: These include JMS messages as well as control messages used by the MQ messaging system to support JMS message delivery.

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

Component	Description/Function
Connection Services	Manages the physical connections between a broker and clients, providing transport for incoming and outgoing messages.
Persistence Manager	Manages the writing of data to persistent storage so that system failure does not result in failure to deliver JMS messages.
Security Manager	Provides authentication services for users requesting connections to a broker and authorization services (access control) for authenticated users.
Monitoring Service	Generates metrics and diagnostic information that can be written to a number of output channels that an administrator can use to monitor and manage a broker.

You can configure these internal components to optimize the performance of the broker, depending on load conditions, application complexity, and so on. The following sections explore more fully the functions performed by the different components and the properties that can be configured to affect their behavior.

Connection Services

An MQ broker supports communication with both MQ application clients and MQ administration clients (see "MQ Administration Tools" on page 96). Each service is specified by its service type and protocol type.

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

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

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

Table 2-2 Connection Services Supported by a Broker

Service Name	Service Type	Protocol Type
jms	NORMAL	tcp
ssljms (Enterprise Edition)	NORMAL	tls (SSL-based security)
httpjms (Enterprise Edition)	NORMAL	http

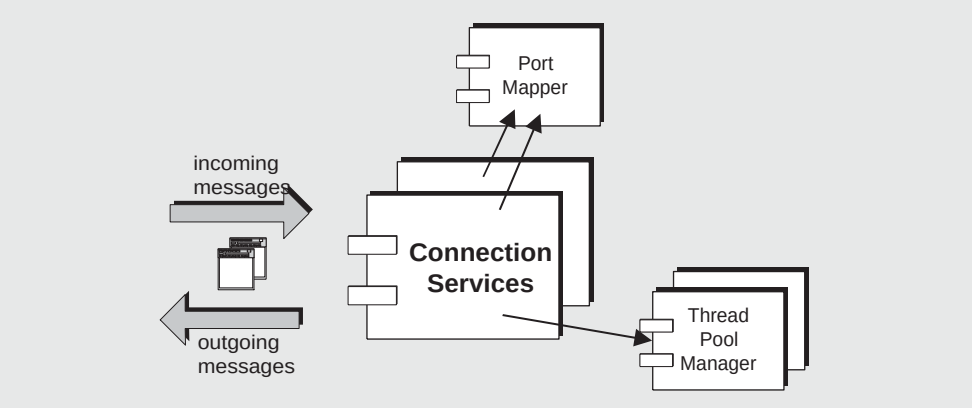
Table 2-2 Connection Services Supported by a Broker *(Continued)*

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

You can configure a broker to run any or all of these connection services. Each connection service is available at a particular port, specified by the broker's host name and a port number. The port can be dynamically allocated or you can specify the port at which a connection service is available.

Each service registers itself with a common Port Mapper but has its own Thread Pool Manager, as shown in Figure 2-3.

Figure 2-3 Connection Services Support



Port Mapper

MQ provides a *Port Mapper* that maps ports to the different connection services. The Port Mapper itself resides at a standard port number, 7676. When a client sets up a connection with the broker, it first contacts the Port Mapper requesting the port number of the connection service it desires.

You can also assign a *static* port number for the jms, ssljms, admin and ssladmin connection services when configuring these connection services, but this is done in special situations (for example, in connections through a firewall) and not generally recommended. The httpjms and httpsjms services are configured using properties described in Table C-1 on page 309 and Table C-3 on page 321, respectively, in Appendix C, "HTTP/HTTPS Support (Enterprise Edition)."

Thread Pool Manager

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

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

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

Shared model (Enterprise Edition) Connections are processed by a shared thread whenever sending or receiving messages. Because each connection does not require dedicated threads, this model increases the number of connections that a connection service (and therefore, a broker) can support. However there is some performance overhead involved in the sharing of threads. The Thread Pool Manager uses a set of distributor threads that monitor connection activity and assign connections to threads as needed. The performance overhead involved in this activity can be minimized by limiting the number of connections monitored by each such distributor thread.

Security

Each connection service supports specific authentication and authorization (access control) features (see "Security Manager" on page 66).

Connection Service Properties

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

Table 2-3 Connection Service Properties

Property Name	Description
`imq.service.activelist`	List of connection services, by name, separated by commas, to be made active at broker startup. Supported services are: `jms`, `ssljms`, `httpjms`, `httpsjms`, `admin`, `ssladmin`. Default: `jms`, `admin`
`imq.ping.interval`	The period between successive attempts of the broker to ping the MQ client runtime across a connection. Default:120 seconds
`imq.hostname`	Specifies the host (hostname or IP address) to which all connection services bind if there is more than one host available (for example, if there is more than one network interface card in a computer). Default: all available IP addresses
`imq.portmapper.port`	Specifies the broker's primary port—the port at which the Port Mapper resides. If you are running more than one broker instance on a host, each must be assigned a unique Port Mapper port. Default: `7676`
`imq.portmapper.hostname`	Specifies the host (hostname or IP address) to which the Port Mapper binds if there is more than one host available (for example, if there is more than one network interface card in a computer). Default: inherits the value of `imq.hostname`
`imq.portmapper.backlog`	Specifies the maximum number of concurrent requests that the Port Mapper can handle before rejecting requests. The property sets the number of requests that can be stored in the operating system backlog waiting to be handled by the port mapper. Default: `50`.
`imq.`*service_name*`.`*protocol_type*[1]`.port`	For jms, ssljms, admin, and ssladmin services only, specifies the port number for the named connection service. Default: `0` (port is dynamically allocated by the Port Mapper) To configure the httpjms and httpsjms connection services, see Appendix C, "HTTP/HTTPS Support (Enterprise Edition)."

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

Property Name	Description
`imq.`*service_name*`.` *protocol_type*[1]`.hostname`	For jms, ssljms, admin, and ssladmin services only, specifies the host (hostname or IP address) to which the named connection service binds if there is more than one host available (for example, if there is more than one network interface card in a computer). Default: inherits the value of `imq.hostname`
`imq.`*service_name*`.` `min_threads`	Specifies the number of threads, which once reached, are maintained in the thread pool for use by the named connection service. Default: Depends on connection service (see Table 5-1 on page 130).
`imq.`*service_name*`.` `max_threads`	Specifies the number of threads beyond which no new threads are added to the thread pool for use by the named connection service. The number must be greater than zero and greater in value than the value of `min_threads`. Default: Depends on connection service (see Table 5-1 on page 130).
`imq.`*service_name*`.` `threadpool_model`	Specifies whether threads are dedicated to connections (`dedicated`) or shared by connections as needed (`shared`) for the named connection service. Shared model (threadpool management) increases the number of connections supported by a broker, but is implemented only for the `jms` and `admin` connection services. Default: Depends on connection service (see Table 5-1 on page 130).
`imq.shared.` `connectionMonitor_limit`	For shared threadpool model only, specifies the maximum number of connections that can be monitored by a distributor thread. (The system allocates enough distributor threads to monitor all connections.) The smaller this value, the faster the system can assign active connections to threads. A value of `-1` means no limit. Default: Depends on operating system (see Table 5-1 on page 130).

1. *protocol_type* is specified in Table 2-2.

Message Router

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

Basic Delivery Mechanisms

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

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

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

Once the Message Router has delivered a message to all its intended consumers it clears the message from memory, and if the message is persistent (see "Reliable Messaging" on page 45), removes it from the broker's persistent data store.

Reliable Delivery: Acknowledgements and Transactions

The delivery mechanism just described becomes more complicated when adding requirements for *reliable* delivery (see "Reliable Messaging" on page 45). There are two aspects involved in reliable delivery: assuring that delivery of messages to and from a broker is successful, and assuring that the broker does not lose messages or delivery information before messages are actually delivered.

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

For example, when a producer sends a JMS message (a payload message as opposed to a control message) to a destination, the broker sends back a control message—a broker acknowledgement—that it received the JMS message. (By default, MQ only does this if the producer specifies the JMS message as persistent.) The producing client uses the broker acknowledgement to guarantee delivery to the destination (see "Message Production" on page 87).

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 subscriptions 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 87). 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 delivers a JMS message a second time, the message is marked with a Redeliver flag. The broker generally re-delivers a JMS message if a client connection closes before the broker receives a client acknowledgement, and a new connection is subsequently opened. For example, if a message consumer of a queue goes off line before acknowledging a message, and another consumer subsequently registers with the queue, the broker will re-deliver the unacknowledged message to the new consumer.

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

The broker tracks transactions, allowing them to be committed or rolled back should they fail. This transaction management also supports local transactions that are part of larger, distributed transactions (see "Distributed Transactions" on page 46). The broker tracks the state of these transactions until they are committed. When a broker starts up it inspects all uncommitted transactions and, by default, rolls back all transactions except those in a `PREPARED` state.

Reliable Delivery: Persistence

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

A producer client can specify that a message be persistent, and in this case, the Message Router will pass the message to a *Persistence Manager* that stores the message in a database or file system (see "Persistence Manager" on page 63) so that the message can be recovered if the broker fails.

Managing Memory Resources and Message Flow

The performance and stability of a broker depends on the system resources available and how efficiently resources such as memory are utilized. In particular, the Message Router could become overwhelmed, using up all its memory resources, when production of messages is much faster than consumption. To prevent this from happening, the Message Router uses three levels of memory protection to keep the system operating as resources become scarce:

Message limits on individual destinations You can set attributes on physical destinations that specify limits on the number of messages and the total memory consumed by messages (see Table 6-10 on page 171), and you can also specify which of four responses are taken by the Message Router when any such limits are reached. The four limit behaviors are:

- slowing message producers
- throwing out the oldest messages in memory
- throwing out the lowest priority messages in memory, according to age of the messages
- rejecting the newest messages

System-wide message limits System-wide message limits constitute a second line of protection. You can specify system-wide limits that apply collectively to all destinations on the system: the total number of messages and the memory consumed by all messages (see Table 2-4 on page 62). If any of the system-wide message limits are reached, the Message Router rejects new messages.

System memory thresholds System memory thresholds are a third line of protection. You can specify thresholds of available system memory at which the broker takes increasingly serious action to prevent memory overload. The action taken depends on the state of memory resources: `green` (plenty of memory is

available), `yellow` (broker memory is running low), `orange` (broker is low on memory), `red` (broker is out of memory). As the broker's memory state progresses from `green` through `yellow` and `orange` to `red`, the broker takes increasingly serious actions of the following types:

- swapping messages out of active memory into persistent storage (see "Persistence Manager" on page 63); non-persistent messages, which normally are not stored, might be swapped out so the system can reclaim memory
- throttling back producers of non-persistent messages, eventually stopping the flow of messages into the broker (persistent message flow is automatically limited by the requirement that each message be acknowledged by the broker)

Both of these measures degrade performance.

If system memory thresholds are reached, then you have not adequately set destination-by-destination message limits and system-wide message limits. In some situations, it is not possible for the thresholds to catch potential memory overloads in time. Hence you should not rely on this feature to control memory resources, but should instead configure destinations individually and collectively to optimize memory resources.

Message Router Properties

System-wide limits and system memory thresholds for managing memory resource are detailed in Table 2-4. (For instructions on setting these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 2-4 Message Router Properties

Property Name	Description
`imq.message.expiration.interval`	Specifies how often reclamation of expired messages occurs, in seconds. Default: `60`
`imq.system.max_count`	Specifies maximum number of messages held by the broker. Additional messages will be rejected. A value of -1 means no limit. Default: -1
`imq.system.max_size`	Specifies maximum total size (in bytes, Kbytes, or Mbytes) of messages held by the broker. Additional messages will be rejected. A value of -1 means no limit. Default: -1
`imq.message.max_size`	Specifies maximum allowed size (in bytes, Kbytes, or Mbytes) of a message body. Any message larger than this will be rejected. A value of -1 means no limit. Default: `70m` (Mbytes)

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

Property Name	Description
`imq.`*resource_state*`.threshold`	Specifies the percent memory utilization at which each memory resource state is triggered. The resource state can have the values `green`, `yellow`, `orange`, and `red`. Defaults: 0, 80, 90, and 98, respectively
`imq.`*resource_state*`.count`	Specifies the maximum number of incoming messages allowed in a batch as each memory resource state is triggered. This limit throttles back message producers as system memory becomes increasingly scarce. Defaults: 5000, 500, 50, and 0, respectively
`imq.transaction.autorollback`	Specifies (`true/false`) whether distributed transactions left in a `PREPARED` state are automatically rolled back when a broker is started up. If `false`, you must manually commit or roll back transactions using `imqcmd` (see "Managing Transactions" on page 180). Default: `false`

Persistence Manager

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

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

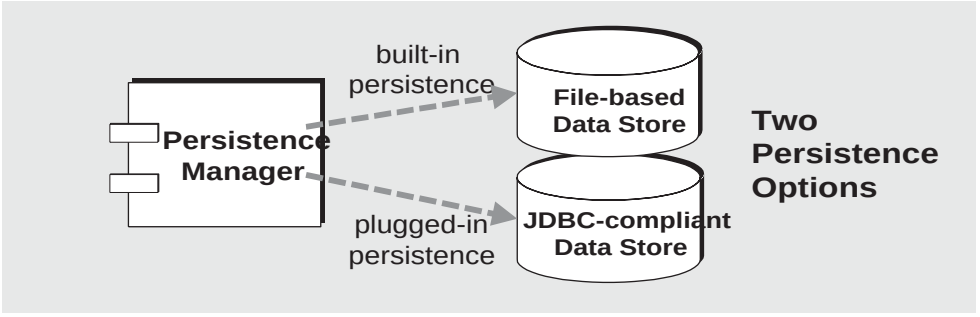
- re-create destinations
- restore the list of durable subscriptions for each topic
- restore the acknowledge list for each message
- reproduce the state of all committed transactions

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

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

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

Figure 2-4 Persistence Manager Support



Built-in persistence

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

The file-based data store is located in a directory identified by the name of the broker instance (*instanceName*) with which the data store is associated (see Appendix A, "Location of MQ Data"):

.../instances/*instanceName*/fs350/

The file-based data store is structured so that persistent messages are stored in a directory according to the destination in which they reside. Most messages are stored in a single file consisting of variable-sized records.

To alleviate fragmentation as messages are added and removed, you can compact the variable-sized record file (see "Compacting Destinations" on page 176). In addition, built-in persistence manager stores messages whose size exceeds a configurable threshold (`imq.persist.file.message.max_record_size`) in their own respective files, rather than in the variable-sized record file. For these individual files, a file pool is maintained so that files can be reused. When a message file is no longer needed, instead of being deleted, it is added to the pool of free files in its destination directory, to be used to store new messages.

You can configure the maximum number of files in the destination file pool (`imq.persist.file.destination.message.filepool.limit`) and specify the percentage of free files in the file pool (the `imq.persist.file.message.filepool.cleanratio`) that are cleaned up—truncated to zero—as opposed to being simply tagged for reuse (not truncated). The higher the percentage of cleaned files, the less disk space—but the more overhead—is required to maintain the file pool. You can also specify whether or not tagged files will be cleaned up at shutdown (`imq.persist.file.message.cleanup`). If the files are cleaned up, they will take up less disk space, but the broker will take longer to shut down.

All other persistent data (destinations, durable subscriptions, and transactions) are stored in their own separate file: all destinations in one file, all durable subscriptions in another, and so on.

To maximize reliability, you can specify (`imq.persist.file.sync.enabled`) that persistence operations synchronize the in-memory state with the physical storage device. This helps eliminate data loss due to system crashes, but at the expense of performance.

Because the data store can contain messages of a sensitive or proprietary nature, it is recommended that the …`instances/`*instanceName*`/fs350/` directory be secured against unauthorized access. For instructions, see "Securing Persistent Data" on page 339.

Plugged-in persistence

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

Persistence Manager Properties

Persistence-related configuration properties are detailed in Table 2-5 on page 66. (For instructions on setting these properties, see Chapter 5, "Starting and Configuring a Broker.")

Except for the first of these properties, all the properties in Table 2-5 pertain only to built-in persistence. Properties pertaining to plugged-in persistence are in Table B-1 on page 298.

Table 2-5 Persistence Manager Properties

Property Name	Description
`imq.persist.store`	Specifies whether the broker is using built-in, file-based (`file`) persistence or plugged-in JDBC-compliant (`jdbc`) persistence. Default: `file`
`imq.persist.file.sync.enabled`	Specifies whether persistence operations synchronize in-memory state with the physical storage device. If `true`, data loss due to system crash is eliminated, but at the expense of performance of persistence operations. Default: `false`
`imq.persist.file.message.max_record_size`	For built-in, file-based persistence, specifies the maximum size of message that will be added to the message storage file, as opposed to being stored in a separate file. Default: `1m` (Mbytes)
`imq.persist.file.destination.message.filepool.limit`	For built-in, file-based persistence, specifies the maximum number of free files available for reuse in the destination 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: `100`
`imq.persist.file.message.filepool.cleanratio`	For built-in, file-based persistence, specifies the percentage of free files in destination file pools that are maintained in a *clean* state (truncated to zero). The higher this value, the more overhead required to clean files during operation, but the less disk space required for the file pool. Default: `0`
`imq.persist.file.message.cleanup`	For built-in, file-based persistence, specifies whether or not the broker cleans up free files in destination file pools on shutdown. A value of false speeds up broker shutdown, but requires more disk space for the file store. Default: `false`

Security Manager

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

The authentication and authorization features depend upon a user repository (see Figure 2-5 on page 68): a file, directory, or database that contains information about the users of the messaging system—their names, passwords, and group memberships. The names and passwords are used to authenticate a user when a connection to a broker is requested. The user names and group memberships are used, in conjunction with an access control file, to authorize operations such as producing or consuming messages for destinations.

MQ administrators populate an MQ-provided user repository (see "Using a Flat-File User Repository" on page 202), or plug a pre-existing LDAP user repository into the Security Manager component (see "Using an LDAP Server for a User Repository" on page 209). The flat-file user repository is easy to use, but is also vulnerable to security attack, and should therefore be used *only* for evaluation and development purposes, while the LDAP user repository is secure and therefore best suited for production purposes.

Authentication

MQ security supports password-based authentication. When a client requests a connection to a broker, the client must submit a user name and password. The Security Manager compares the name and password submitted by the client to those stored in the user repository. On transmitting the password from client to broker, the passwords are encoded using either base 64 encoding or message digest (MD5). For more secure transmission, see "Encryption (Enterprise Edition)" on page 68. You can separately configure the type of encoding used by each connection service or set the encoding on a broker-wide basis.

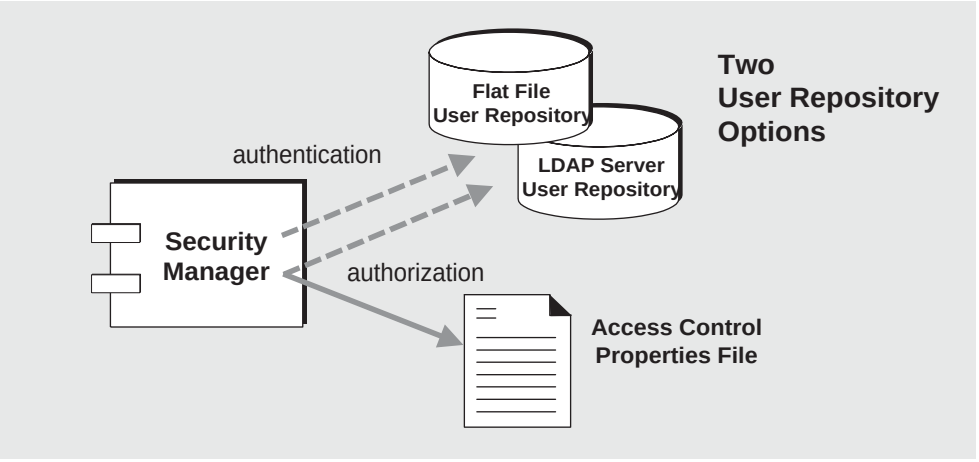
Authorization

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

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

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

Figure 2-5 Security Manager Support



The default access control properties file explicitly references only one group: *admin* (see "Groups" on page 205). A user in the *admin* group has admin service connection permission. The admin service lets the user perform administrative functions such as creating destinations, and monitoring and controlling a broker. A user in any other group you define cannot, by default, get an admin service connection.

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

In addition, if the broker is configured to allow auto-creation of destinations (see "Auto-Created (vs. Admin-Created) Destinations" on page 78), you can control for whom the broker can auto-create destinations by editing the access control properties file.

Encryption (Enterprise Edition)

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

To use an MQ SSL-based connection service, you generate a private key/public key pair using the Key Tool utility (`imqkeytool`). This utility embeds the public key in a self-signed certificate and places it in an MQ keystore. The MQ keystore is, itself, password protected; to unlock it, you have to provide a keystore password at startup time. See "Encryption: Working With an SSL-based Service (Enterprise Edition)" on page 218.

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

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

Table 2-6 Security Manager Properties

Property Name	Description
`imq.authentication.type`	Specifies whether the password should be passed in base 64 coding (`basic`) or as a MD5 digest (`digest`). Sets encoding for all connection services supported by a broker. Default: `digest`
`imq.`*service_name*`.authentication.type`	Specifies whether the password should be passed in base 64 coding (`basic`) or as a MD5 digest (`digest`). Sets encoding for named connection service, overriding any broker-wide setting. Default: inherits value of `imq.authentication.type`
`imq.authentication.basic.user_repository`	Specifies (for base 64 coding) the type of user repository used for authentication, either file-based (`file`) or LDAP (`ldap`). For additional LDAP properties, see Table 8-5 on page 210. Default: `file`
`imq.authentication.client.response.timeout`	Specifies the time (in seconds) the system will wait for a client to respond to an authentication request from the broker. Default: `180` (seconds)
`imq.accesscontrol.enabled`	Sets access control (`true`/`false`) for all connection services supported by a broker. Indicates whether system will check if an authenticated user has permission to use a connection service or to perform specific MQ operations with respect to specific destinations, as specified in the access control properties file. Default: `true`

Table 2-6 Security Manager Properties *(Continued)*

Property Name	Description
`imq.`*service_name*`.accesscontrol.enabled`	Sets access control (`true`/`false`) for named connection service, overriding broker-wide setting. Indicates whether system will check if an authenticated user has permission to use the named connection service or to perform specific MQ operations with respect to specific destinations, as specified in the access control properties file. Default: inherits the value of `imq.accesscontrol.enabled`
`imq.accesscontrol.file.filename`	Specifies the name of an access control properties file for all connection services supported by a broker instance. The file name specifies a relative file path to the access control directory (see Appendix A, "Location of MQ Data"). Default: `accesscontrol.properties`
`imq.`*service_name*`.accesscontrol.file.filename`	Specifies the name of an access control properties file for a named connection service of a broker instance. The file name specifies a relative file path to the access control directory (see Appendix A, "Location of MQ Data"). Default: inherits the value of `imq.accesscontrol.file.filename`.
`imq.passfile.enabled`	Specifies (true/false) if user passwords (for SSL, LDAP, JDBC™) for secure communications are specified in a passfile. Default: `false`
`imq.passfile.dirpath`	Specifies the path to the directory containing the passfile (depends on operating system). Default: see Appendix A, "Location of MQ Data"
`imq.passfile.name`	Specifies the name of the passfile. Default: `passfile`
`imq.keystore.`*property_name*	For SSL-based services: specifies security properties relating to the SSL keystore. See Table 8-8 on page 220.

Monitoring Service

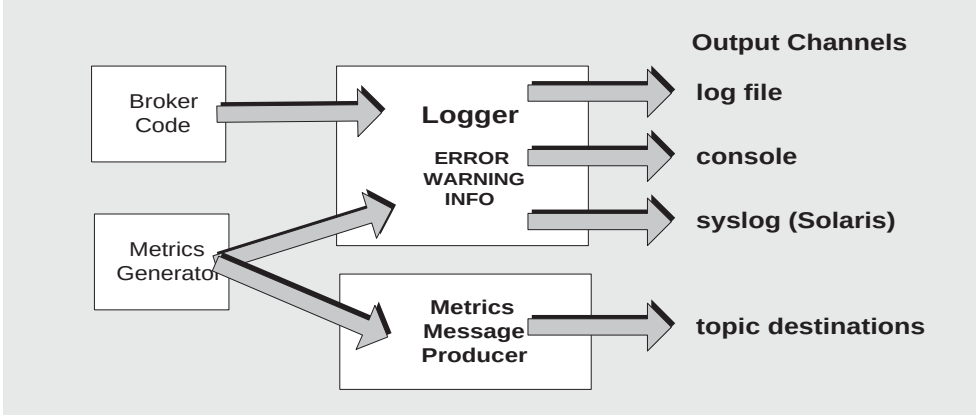
The broker includes a number of components for monitoring and diagnosing its operation. Among these are the following:

- Components that generate data (broker code that logs events and a metrics generator)
- A Logger component (see "Logger") that writes out information through a number of output channels

- A message producer that sends JMS messages containing metrics information to topic destinations for consumption by JMS monitoring clients.

The general scheme is illustrated in Figure 2-6.

Figure 2-6 Monitoring Service Support



Metrics Generator

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

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

Logger

The MQ logger takes information generated by broker code and a metrics generator and writes that information to a number of output channels: to standard output (the console), to a log file, and, on Solaris™ platforms, to the `syslog` daemon process.

You can specify the type of information gathered by the logger as well as the type written to each of the output channels.

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

Table 2-7 Logging Categories

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

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

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

In the case of a log file, you can 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 written to a directory identified by the name of the broker instance (*instanceName*) with which the log file is associated (see Appendix A, "Location of MQ Data"):

.../instances/*instanceName*/log/

An archive of the nine most recent log files is retained as new rollover log files are created.

For information on configuring the Logger, see Table 2-9 on page 74 and "Changing the Logger Configuration" on page 148.

Metrics Message Producer (Enterprise Edition)

The Message Producer component receives information from the Metrics Generator component at regular intervals, and writes the information into messages, which it then sends to one of a number of metric topic destinations, depending on the type of metrics information contained in the message.

There are five metrics topic destinations, whose names are shown in Table 2-8, along with the type of metrics messages delivered to each destination.

Table 2-8 Metrics Topic Destinations

Topic Destination Name	Type of Metrics Messages
mq.metrics.broker	Broker metrics
mq.metrics.jvm	Java Virtual Machine metrics
mq.metrics.destination_list	List of destinations and their types
mq.metrics.destination.queue.*monitoredDestinationName*	Destination metrics for queue of specified name
mq.metrics.destination.topic.*monitoredDestinationName*	Destination metrics for topic of specified name

MQ clients subscribed to these metric topic destinations consume the messages in the destinations and process the metrics information contained in the messages. For example, a client can subscribe to the `mq.metrics.broker` destination to receive and process information such as the total number of messages in the broker.

The Metrics Message Producer is an internal MQ client that creates messages (of type MapMessage) that contain name-value pairs corresponding to metrics data. These messages are produced only if there is one or more subscribers to the corresponding metrics topic destination.

The messages produced by the Metrics Message Producer are of type MapMessage; they consist of a number of name/value pairs, depending on the type of metrics they contain. Each name/value pair corresponds to a metric quantity and its value. For example, broker metrics messages contain values for about a dozen metric quantities, including the number of messages that have flowed into and out of the broker, the size of these messages, the number and size of messages currently in memory, and so forth. For details of the metrics quantities reported in each type of metrics message, see the MQ *Java Client Developer's Guide*, which explains how to write an MQ client for consuming metrics messages.

In addition to the metrics information contained in the body of a metrics message, the header of each message has two properties: one which specifies the metrics message type and one which records a timestamp. These header properties can be used by MQ client applications to extract data from metrics messages and record their corresponding timestamps.

Monitoring Service Properties

The configurable properties for setting the generation, logging, and metrics message production of information by the broker are shown in Table 2-9. (For instructions on configuring these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 2-9 Monitoring Service Properties

Property Name	Description
`imq.metrics.enabled`	Specifies (`true/false`) whether metrics information is being written to the logger. Does not affect production of metrics messages (see `imq.metrics.topic.enabled`). Default: `true`
`imq.metrics.interval`	If metrics logging is enabled (`imq.metrics.enabled=true`), specifies the time interval, in seconds, at which metrics information is written to the Logger. A value of `-1` means never. Does not affect time interval for production of metrics messages (see `imq.metrics.topic.interval`). Default: `-1`
`imq.log.level`	Specifies the Logger level: the categories of output that can be written to an output channel. Includes the specified category and all higher level categories as well. Values, from high to low, are: `ERROR`, `WARNING`, `INFO`. Default: `INFO`
`imq.log.file.output`	Specifies which categories of logging information are written to the log file. Allowed values are: any set of logging categories separated by vertical bars (|), or `ALL`, or `NONE`. Default: `ALL`
`imq.log.file.dirpath`	Specifies the path to the directory containing the log file (depends on operating system). Default: see Appendix A, "Location of MQ Data"
`imq.log.file.filename`	Specifies the name of the log file. Default: `log.txt`
`imq.log.file.rolloverbytes`	Specifies the size, in bytes, of log file at which output rolls over to a new log file. A value of `-1` means no rollover based on file size. Default: `-1`
`imq.log.file.rolloversecs`	Specifies the age, in seconds, of log file at which output rolls over to a new log file. A value of `-1` means no rollover based on age of file. Default: `604800` (one week)

Table 2-9 Monitoring Service Properties *(Continued)*

Property Name	Description
imq.log.console.output	Specifies which categories of logging information are written to the console. Allowed values are any set of logging categories separated by vertical bars (|), or ALL, or NONE. Default: ERROR| WARNING
imq.log.console.stream	Specifies whether console output is written to stdout (OUT) or stderr (ERR). Default: ERR
imq.log.syslog.facility	(Solaris only) Specifies what syslog facility the MQ broker should log as. Values mirror those listed in the syslog(3C) man page. Appropriate values for use with MQ are: LOG_USER, LOG_DAEMON, and LOG_LOCAL0 through LOG_LOCAL7. Default: LOG_DAEMON
imq.log.syslog.logpid	(Solaris only) Specifies (true/false) whether to log the broker process ID with the message or not. Default: true
imq.log.syslog.logconsole	(Solaris only) Specifies (true/false) whether to write messages to the system console if they cannot be sent to syslog. Default: false
imq.log.syslog.identity	(Solaris only) Specifies the identity string that should be prepended to every message logged to syslog. Default: imqbrokerd_ followed by the broker instance name.
imq.log.syslog.output	(Solaris only) Specifies which categories of logging information are written to syslogd(1M). Allowed values are any logging categories separated by vertical bars (|), or ALL, or NONE. Default: ERROR
imq.log.timezone	Specifies the time zone for log time stamps. The identifiers are the same as those used by java.util.TimeZone.getTimeZone(). For example: GMT, America/LosAngeles, Europe/Rome, Asia/Tokyo. Default: local time zone
imq.metrics.topic.enabled	Specifies (true/false) whether metrics message production is enabled. If false, an attempt to subscribe to a metric topic destination will throw a client-side exception. Default: true

Table 2-9 Monitoring Service Properties *(Continued)*

Property Name	Description
`imq.metrics.topic.interval`	Specifies the time interval, in seconds, at which metrics messages are produced (sent to metric topic destinations). Default: `60`
`imq.metrics.topic.persist`	Specifies (`true/false`) whether metrics messages are persistent or not. Default: `false`
`imq.metrics.topic.timetolive`	Specifies the lifetime, in seconds, of metrics messages sent to metric topic destinations. Default: `300`

Physical Destinations

MQ messaging is premised on a two-phase delivery of messages: first, delivery of a message from a producer client to a destination on the broker, and second, delivery of the message from the destination on the broker to one or more consumer clients. There are two types of destinations (see "Programming Domains" on page 43): queues (point-to-point delivery model) and topics (publish/subscribe delivery model). These destinations represent locations in a broker's physical memory where incoming messages are marshaled before being routed to consumer clients.

You create physical destinations using MQ administration tools (see "Getting Connection Information" on page 167). Destinations can also be automatically created as described in "Auto-Created (vs. Admin-Created) Destinations" on page 78.

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 message producers, they are queued and delivered to a single message consumer.

Queue Delivery to Multiple Consumers

While any message in a queue destination is delivered to only a single consumer, MQ allows multiple consumers to register with a queue. The broker routes incoming messages to the different consumers, balancing the load among them.

The implementation of queue delivery to multiple consumers uses a configurable load-balancing approach based on the following queue destination attributes:

- `maxNumActiveConsumers`: Specifies the number of consumers—one or many—that are active in load-balanced queue delivery
- `maxNumBackupConsumers`: Specifies the number of backup consumers—none or many—that can take the place of active consumers should any active consumers fail.

New consumers will be rejected if the number of consumers exceeds the sum of these two attributes. (MQ Platform Edition supports up to 3 consumers per queue—two active and one backup—and MQ Enterprise Edition supports an unlimited number.)

The load-balancing mechanism takes into account the message consumption rate of different consumers. It works like this:

Messages in a queue destination are routed to newly available active consumers (in the order in which they registered with the queue) in batches of a configurable size (the queue destination's `consumerFlowLimit` attribute). Once these messages have been delivered, additional messages arriving in the queue are routed in batches to consumers as consumers become available (that is, when the consumers have consumed a configurable percentage of messages previously delivered to them). The dispatch rate to each consumer depends on the consumer's current capacity and message processing rate.

When the rate of message production is slow, the broker might dispatch messages unevenly among active consumers. If you have more active consumers than necessary, some may never receive messages.

If an active consumer fails, then the first backup consumer is made active, and takes over the work of the failed consumer. When a queue destination has more than one active consumer, no guarantee can be made about the order in which messages are consumed.

In a broker cluster environment, delivery to multiple consumers can be set to prioritize local consumers. A queue destination attribute, `localDeliveryPreferred`, lets you specify that messages be delivered to remote consumers only if there are no consumers on a producer's home broker—that is, the broker to which the producer sent its messages (the local broker). This lets you increase performance in situations where routing to remote consumers (through *their* home brokers) might cause slowdowns in throughput. (This attribute requires that the destination's scope not be restricted to local-only delivery—see Table 6-10 on page 171.)

Memory Considerations

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 could be rejected). To allow for flexibility, based on the load demands of each queue, you can set physical properties when creating a queue: maximum number of messages in queue, maximum memory allocated for messages in queue, and maximum size of any message in queue (see Table 6-10 on page 171).

Topic Destinations

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

Messages do not normally remain in a topic destination for an extended period of time, so memory resources are not normally a big issue. However, you can configure the maximum size allowed for any message received by the destination (see Table 6-10 on page 171).

Auto-Created (vs. Admin-Created) Destinations

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

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

MQ supports this *auto-create* capability. When auto-creation is enabled, a broker automatically creates a destination whenever a MessageConsumer or MessageProducer attempts to access a non-existent destination. (The user of the client application must have auto-create privileges—see "Destination Auto-Create Access Control" on page 217).

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 auto-created destination is automatically destroyed by the broker when it is no longer being used: that is, when it no longer has message consumer clients and no longer contains any messages. If a broker is restarted, it will only re-create auto-created destinations if they contain persistent messages.

You can configure an MQ message server to enable or disable the auto-create capability using the properties shown in Table 2-10. (For instructions on configuring these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 2-10 Auto-create Configuration Properties

Property Name	Description
`imq.autocreate.topic`	Specifies (`true/false`) whether a broker is allowed to auto-create a topic destination. Default: `true`
`imq.autocreate.queue`	Specifies (`true/false`) whether a broker is allowed to auto-create a queue destination. Default: `true`
`imq.autocreate.destination.maxNumMsgs`	Specifies maximum number of unconsumed messages allowed in an auto-created destination. Default: `100,000`
`imq.autocreate.destination.maxTotalMsgBytes`	Specifies the maximum total amount of memory (in bytes) allowed for unconsumed messages in the destination. Default: `10m` (megabytes)
`imq.autocreate.destination.limitBehavior`	Specifies how the broker responds when a memory-limit threshold is reached. Values are: `FLOW_CONTROL` — slows down producers `REMOVE_OLDEST` — throws out oldest messages `REMOVE_LOW_PRIORITY` — throws out lowest priority messages according to age of the messages `REJECT_NEWEST` — rejects the newest messages Default: `REJECT_NEWEST`
`imq.autocreate.destination.maxBytesPerMsg`	Specifies maximum size (in bytes) of any single message allowed in an auto-created destination. Default: `10k` (`10,240`)
`imq.autocreate.destination.maxNumProducers`	Specifies maximum number of producers allowed for the destination. When this limit is reached, no new producers can be created. Default: `100`

Table 2-10 Auto-create Configuration Properties *(Continued)*

Property Name	Description
`imq.autocreate.destination.isLocalOnly`	Applies only to broker clusters. Specifies that a destination is not replicated on other brokers, and is therefore limited to delivering messages only to local consumers (consumers connected to the broker on which the destination is created). This attribute cannot be updated once the destination has been created. Default: `false`
`imq.autocreate.queue.maxNumActiveConsumers`	Specifies the maximum number of consumers that can be active in load-balanced delivery from an auto-created queue destination. A value of `-1` means an unlimited number. Default: `1`
`imq.autocreate.queue.maxNumBackupConsumers`	Specifies the maximum number of backup consumers that can take the place of active consumers if any fail during load-balanced delivery from an auto-created queue destination. A value of `-1` means an unlimited number. Default: `0`
`imq.autocreate.queue.consumerFlowLimit`	Specifies the maximum number of messages that will be delivered to a consumer in a single batch. In load-balanced queue delivery, this is the initial number of queued messages routed to active consumers before load-balancing commences (see "Queue Delivery to Multiple Consumers" on page 76). This limit can be overridden by a lower value set for the destination's consumers on their respective connections (see information on Connection Factory attributes in the MQ *Java Client Developer's Guide*). A value of `-1` means an unlimited number. Default: `1000`
`imq.autocreate.topic.consumerFlowLimit`	Specifies the maximum number of messages that will be delivered to a consumer in a single batch. A value of `-1` means an unlimited number. Default: `1,000`
`imq.autocreate.queue.localDeliveryPreferred`	Applies only to load-balanced queue delivery in broker clusters. Specifies that messages be delivered to remote consumers only if there are no consumers on the local broker. Requires that the auto-created destination not be restricted to local-only delivery (`isLocalOnly = false`). Default: `false`

Temporary Destinations

Temporary destinations are explicitly created and destroyed (using the JMS API) by clients 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, however, they are visible to MQ administration tools (see Table 6-9 on page 168).

Multi-Broker Clusters (Enterprise Edition)

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

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

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

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. While MQ allows for a failed connection to be re-established with a different broker in a cluster, state information is lost. Therefore, one broker in a cluster cannot be used as an automatic backup for another that fails.

In other words, MQ does not presently support a high availability message server. However, you can use Sun Cluster software and highly available databases to provide for broker failover. You can also design a messaging application to use multiple brokers to implement a customized failover solution.

Information on configuring and managing a broker cluster is provided in "Working With Clusters (Enterprise Edition)" on page 140.

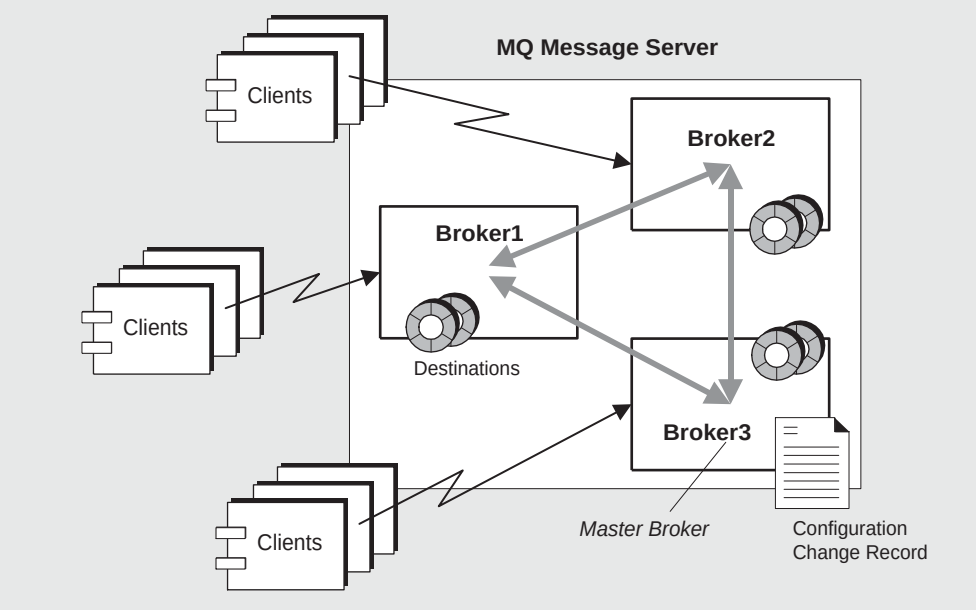
The following sections explain the architecture and internal functioning of MQ broker clusters.

Multi-Broker Architecture

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

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

Figure 2-7 Multi-Broker (Cluster) Architecture



Message Delivery

In a multi-broker configuration, each destination is replicated on all of the brokers in a cluster. (With some exceptions, destination attributes in a cluster environment generally apply *collectively* to all instances of the destination, that is, to the cluster as a whole, rather than to the individual destination instances. Also, destinations for which the `isLocalOnly` attribute is set to `true` are not replicated in a cluster. For more information, see the description of destination attributes, Table 6-10 on page 171.)

Each broker knows about message consumers that are registered for destinations on 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.

To minimize traffic between the brokers in a cluster, delivery of messages (from destination to client runtime) is regulated by flow control mechanisms on consumer connections. In this way, messages are sent from one broker to another only when they are to be delivered to a consumer connected to the target broker, thereby avoiding the passing of unnecessary messages from broker to broker. Also, in some cases—for example, in the case of queue delivery to multiple consumers—you can minimize broker to broker traffic by specifying that delivery to local consumers have priority over delivery to remote consumers (see the `localDeliveryPreferred` queue destination attribute, Table 6-10 on page 171).

In situations in which secure, encrypted message delivery between client and message server is required, a cluster can be configured to also secure delivery of messages between brokers in a cluster (see "Secure Inter-Broker Connections" on page 143).

Cluster Synchronization

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

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

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

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

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

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

In a mission-critical application environment it is a good idea to make a periodic backup of the configuration change record to guard against accidental corruption of the record and safeguard against Master Broker failure. You can do this using the `-backup` option of the `imqbrokerd` command (see Table 5-2 on page 136), 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 5-3 on page 141) to designate a new Master Broker, and restarting the new Master Broker using the `-restore` option.

Using Clusters in Development Environments

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

Under test situations, destinations are generally auto-created (see "Auto-Created (vs. Admin-Created) Destinations" on page 78) and durable subscriptions to these destinations are created and destroyed by the applications being tested. These changes in destinations and durable subscriptions will be propagated throughout the cluster. However, if you reconfigure the environment to use a Master Broker, MQ will re-impose the requirement that the Master Broker be running for changes to be made in destinations and durable subscriptions, and for these changes to be propagated throughout the cluster.

Cluster Configuration Properties

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

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

You can also duplicate cluster 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.

For more information on cluster configuration properties, see "Working With Clusters (Enterprise Edition)" on page 140.

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

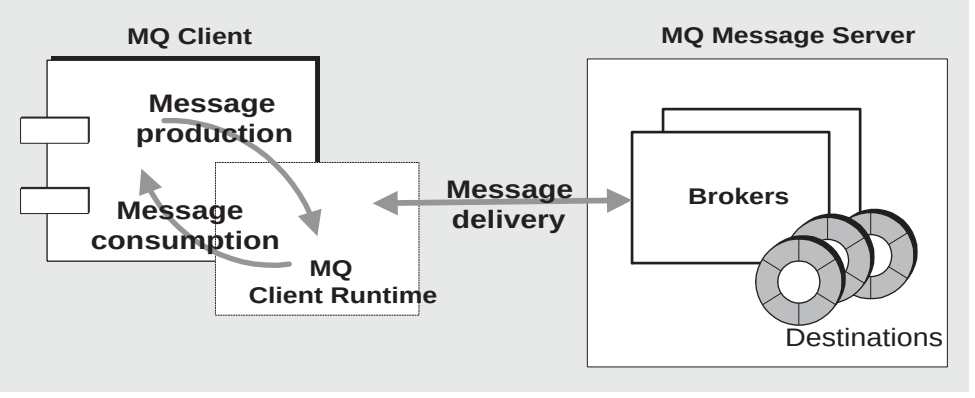
MQ Client Runtime

The MQ client runtime provides client applications with an interface to the MQ message service—it supplies Java client applications with all the JMS programming objects introduced in "JMS Programming Model" on page 38 and C client applications with the corresponding C interfaces. The MQ client runtime supports all operations needed for clients to send messages to destinations and to receive messages from such destinations.

This section provides a high level description of how the MQ client runtime works. Factors that affect client application design and performance of the Java client runtime and the C client runtime are discussed in the MQ *Java Client Developer's Guide* and the MQ *C Client Developer's Guide*, respectively.

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

Figure 2-8 Messaging Operations



Message Production

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

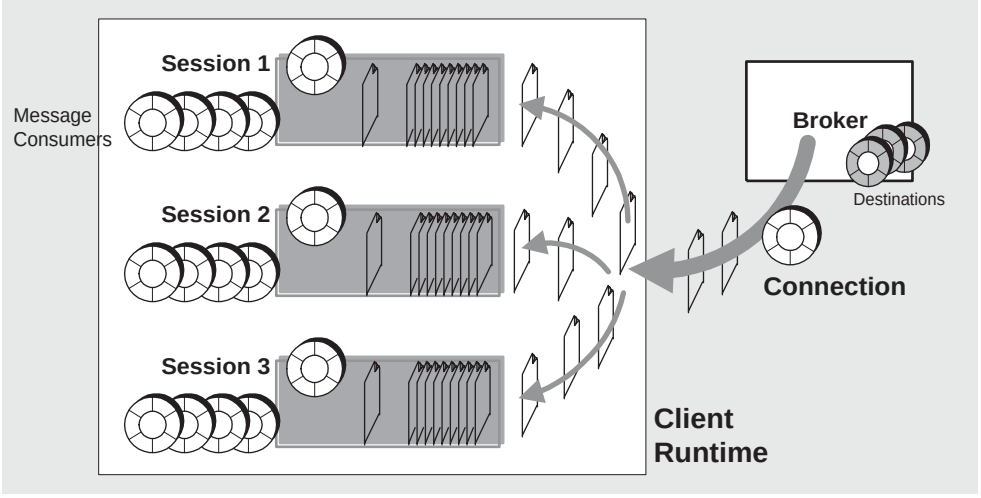
Message Consumption

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

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

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

Figure 2-9 Message Delivery to MQ Client Runtime



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

MQ Administered Objects

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

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

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

- You can control the behavior of connections by requiring client applications to access pre-configured `ConnectionFactory` objects (see "Connection Factory Administered Object Attributes" on page 187).
- You can control the proliferation of physical destinations by requiring client applications to access pre-configured `Destination` objects that correspond to existing physical destinations. (You also have to disable the brokers's auto-create capability—see "Auto-Created (vs. Admin-Created) Destinations" on page 78).
- You can control MQ message server resources by overriding message header values set by client applications (see "Connection Factory Administered Object Attributes" on page 187).

This arrangement therefore gives you, as an MQ administrator, control over message server configuration details, and at the same time allows client applications to be provider-independent: they do not have to know about provider-specific syntax and object naming conventions (see "JMS Provider Independence" on page 43) or provider-specific configuration properties.

You create administered objects using MQ administration tools, as described in Chapter 7, "Managing Administered Objects." When creating an administered object, you can specify that it be read only—that is, client applications are prevented from changing MQ-specific configuration values you have set when creating the object. In other words, client code cannot set attribute values on read-only administered objects, nor can you override these values using client application startup options, as described in "Overriding Attribute Values at Client Startup" on page 92.

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

Connection Factory Administered Objects

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

If you wish to support distributed transactions (see "Local Transactions" on page 46), you need to use a special `XAConnectionFactory` object that supports distributed transactions.

To create a `ConnectionFactory` administered object, see "Adding a Connection Factory" on page 195.

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

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

Each of these categories and its corresponding attributes is discussed in some detail in the MQ *Java Client Developer's Guide*. While you, as an MQ administrator, might be called upon to adjust the values of these attributes, it is normally an application developer who decides which attributes need adjustment to tune the performance of client applications. Table 7-3 on page 187 presents an alphabetical summary of the attributes.

Destination Administered Objects

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

To create a `Destination` administered object, see "Adding a Topic or Queue" on page 196.

Table 2-11 Destination Attributes

Attribute/property name	Description
`imqDestinationName`	Specifies the provider-specific name of the physical destination. You specify this name when you create a physical destination. Destination names must contain only alphanumeric characters (no spaces) and can begin with an alphabetic character or the characters "_" and "$". They cannot begin with the character string "`mq.`" Default: `Untitled_Destination_Object`
`imqDestinationDescription`	Specifies information useful in managing the object. Default: `A Description for the Destination Object`

Overriding Attribute Values at Client Startup

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

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

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

where *attribute* corresponds to any of the `ConnectionFactory` administered object attributes documented in "Connection Factory Administered Object Attributes" on page 187.

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

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

Chapter 3

MQ Administration Tasks and Tools

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

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

MQ Administration Tasks

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

Development Environments

In a development environment, the work focuses on programming MQ client applications. The MQ message server is needed principally for testing. In a development environment, the emphasis is on flexibility, and you typically adopt the following practices:

- You want minimal administration—consisting mostly of starting up a broker for developers to use in testing.
- You use default implementations of the data store (built-in file-based persistence), user repository (file-based user repository), access control properties file, and object store (file-system store). These default implementation are usually adequate for developmental testing.
- If you are performing multi-broker testing, you probably would not use a Master Broker.

- You generally use auto-created destinations rather than explicitly creating destinations
- You might instantiate administered objects in client code rather than use centrally-managed administered objects.

Production Environments

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

Setup Operations

➤ **To Set Up a Production Environment**

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

- **Administrator security** (protected use of administration tools):
  - Make sure admin connection service is activated (see Table 2-3 on page 57)
  - Authorization: allow access to admin connection service for a specific individual or admin group (see "Connection Access Control" on page 215)
  - If authorization is for a group, make sure the administrator belongs to the admin group.
    - File-based user repository: has a default `admin` group. Make sure administrator is in `admin` group, or if using the default admin user, change the admin password (see "Changing the Default Administrator Password" on page 208).
    - LDAP user repository: make sure administrator is in `admin` group
- **General security** (see Chapter 8, "Managing Security"):
  - Authentication: 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.)
  - Authorization: modify access settings in the access control properties file

- Encryption: set up SSL-based connection services

- **Administered objects** (see Chapter 7, "Managing Administered Objects"):
  - configure or set up an LDAP object store
  - create ConnectionFactory and destination administered objects
- **Broker clusters** (see "Working With Clusters (Enterprise Edition)" on page 140):
  - 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 B, "Setting Up Plugged-in Persistence")
- **Memory management**: set destination attributes so that the number of messages and the amount of memory allocated for messages fit within available broker memory resources (see Table 6-10 on page 171)

Maintenance Operations

➤ **To Set Up a Production Environment**

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

- **Application management**:
  - disable the broker's auto-create capability (see Table 2-10 on page 79)
  - create physical destinations on behalf of applications (see "Creating Destinations" on page 170)
  - set user access to destinations (see "Authorizing Users: the Access Control Properties File" on page 212)
  - monitor and manage destinations (see "Managing Destinations" on page 168)
  - monitor and manage durable subscriptions (see "Managing Durable Subscriptions" on page 179)
  - monitor and manage transactions (see "Managing Transactions" on page 180)

- **Broker administration and tuning**:
  - use broker metrics to tune and reconfigure the broker (see Chapter 9, "Analyzing and Tuning an MQ Message Service" on page 225)
  - manage broker memory resources (see Chapter 9, "Analyzing and Tuning an MQ Message Service" on page 225)
  - add brokers to clusters to balance loads (see "Working With Clusters (Enterprise Edition)" on page 140)
  - recover failed brokers (see "Starting a Broker" on page 134)
- **Managing administered objects:**
  - create additional ConnectionFactory and destination administered objects as needed (see "Adding and Deleting Administered Objects" on page 195)
  - adjust ConnectionFactory attribute values to improve performance and throughput (see "Connection Factory Administered Object Attributes" on page 187 and "Updating Administered Objects" on page 200)

MQ Administration Tools

MQ administration tools fall into two categories: command line utilities and a graphical user interface (GUI) Administration Console (`imqadmin`). The Console combines the capabilities of two command line utilities: the Command utility (`imqcmd`) and the Object Manager utility (`imqobjmgr`). You can use the Console (and these two command line utilities) to manage a broker remotely and to manage MQ administered objects. The other command line utilities (`imqbrokerd`, `imqusermgr`, `imqdbmgr`, and `imqkeytool`) must be run on the same host as their associated broker, as shown in Figure 3-1 on page 97.

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

The Administration Console

You can use the administration console to do the following:

- Connect to a broker and manage it.
- Create and manage physical destinations on the broker

- Connect to an object store
- Add administered objects to the object store and manage them.

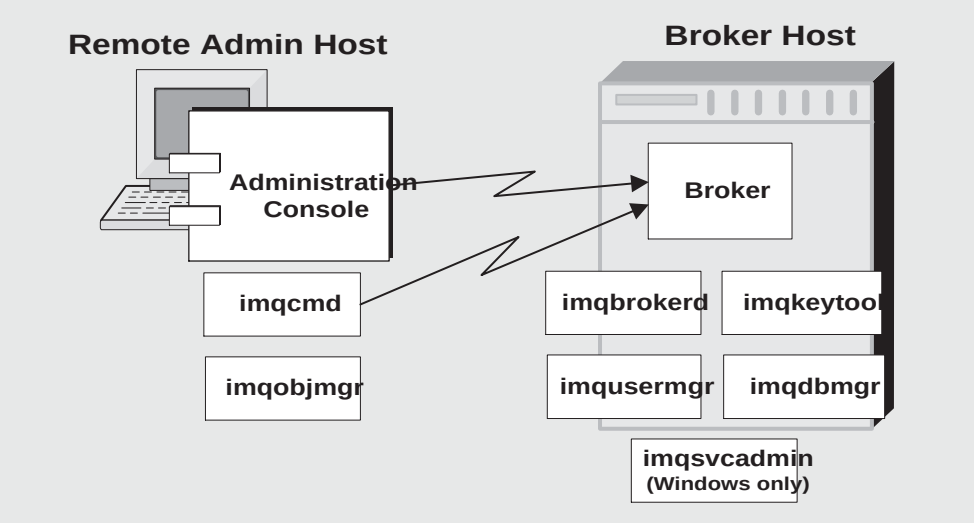
There are some tasks that you cannot use the Administration Console to perform; chief among these are starting up a broker, creating broker clusters, configuring more specialized properties of a broker and physical destinations, and managing a user database.

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

Summary of Command Line Utilities

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

Figure 3-1 Local and Remote Administration Utilities



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

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

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

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

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

Key Tool (`imqkeytool`) You use the Key Tool utility to generate self-signed certificates used for SSL authentication. The `imqkeytool` command is documented in Chapter 8, "Managing Security" and in Appendix C, "HTTP/HTTPS Support (Enterprise Edition)."

Database Manager (`imqdbmgr`) You use the Database Manager utility to create and manage a JDBC-compliant database used for persistent storage. The `imqdbmgr` command is documented in Appendix B, "Setting Up Plugged-in Persistence."

Service Administrator (`imqsvcadmin`) You use the Service Administrator utility to install, query, and remove the broker as a Windows service. The `imqsvcadmin` command is documented in Appendix D, "Using a Broker as a Windows Service."

Command Line Syntax

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

All the command line utilities share the following command syntax:

Utility_Name [*subcommand*] [*argument*] [[-*option_name* [-*option_argument*]]...]

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

There are four important things to remember:

- Specify options *after* subcommands (and arguments, if the utility accepts both types of operands).
- If an argument contains a space, enclose the whole argument in quotation marks. It is generally safest to enclose an attribute-value pair in quotes.
- If you specify the -v (version) or the -h/-H (help) options on a command line, nothing else on that command line is executed. See Table 3-1 on page 100 for a description of common options.
- Separate the subcommand, arguments, options, and option arguments with spaces.

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

```
imqbrokerd
```

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

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

Common Command Line Options

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

Table 3-1 Common MQ Command Line Options

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

Chapter 4

Administration Console Tutorial

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

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

The tutorial is designed to set up the destinations and administered objects needed to run a simple JMS-compliant application, HelloWorldMessageJNDI, which is available in the `helloworld` subdirectory of the example applications /demo directory (see Appendix A, "Location of MQ Data"). In the last part of the tutorial you run this application.

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

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

- Configuring certain physical destination properties

  Some physical destination properties cannot be configured using the Administration Console. These can be configured as described in the section titled "Managing Destinations" on page 168.

- Creating broker clusters

  See "Working With Clusters (Enterprise Edition)" on page 140 for more information.

- Managing a user database

  See "Authenticating Users" on page 202 for more information.

Getting Ready

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

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

Starting the Administration Console

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

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

  Administered objects allow you to manage the messaging needs of JMS-compliant applications. For more information, see "MQ Administered Objects" on page 88.

➤ To Start the Administration Console

1. Choose Start > Programs > Sun ONE Message Queue 3.5 > Administration.

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

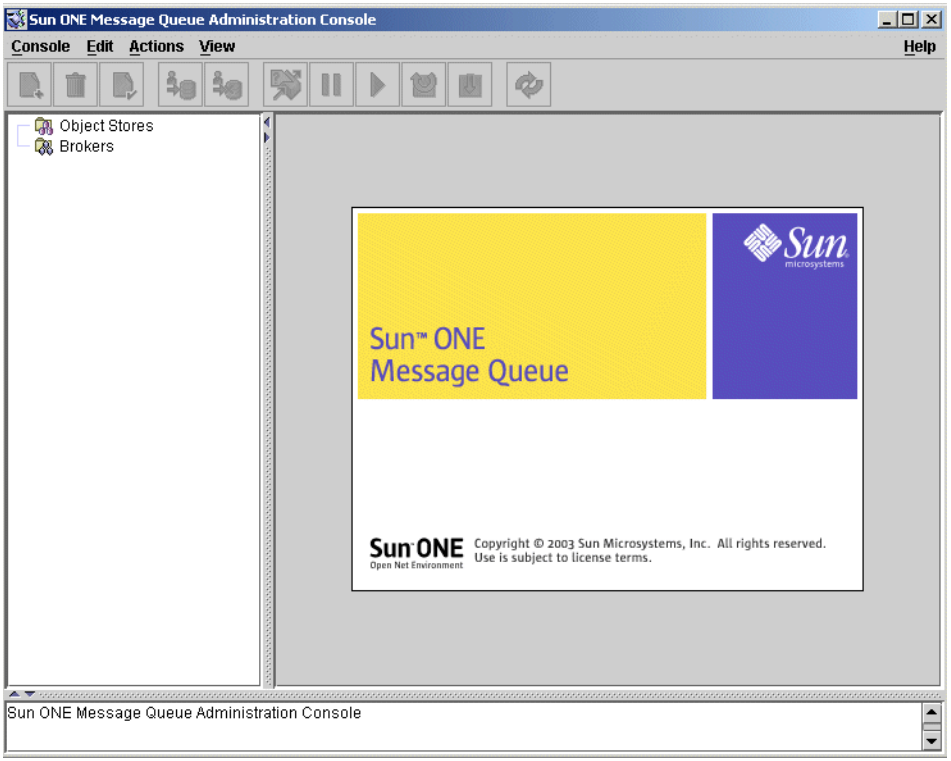
   Non-Windows users: enter the following command at the command prompt:

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

   `/opt/imq/bin/imqadmin` (on Linux)

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

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



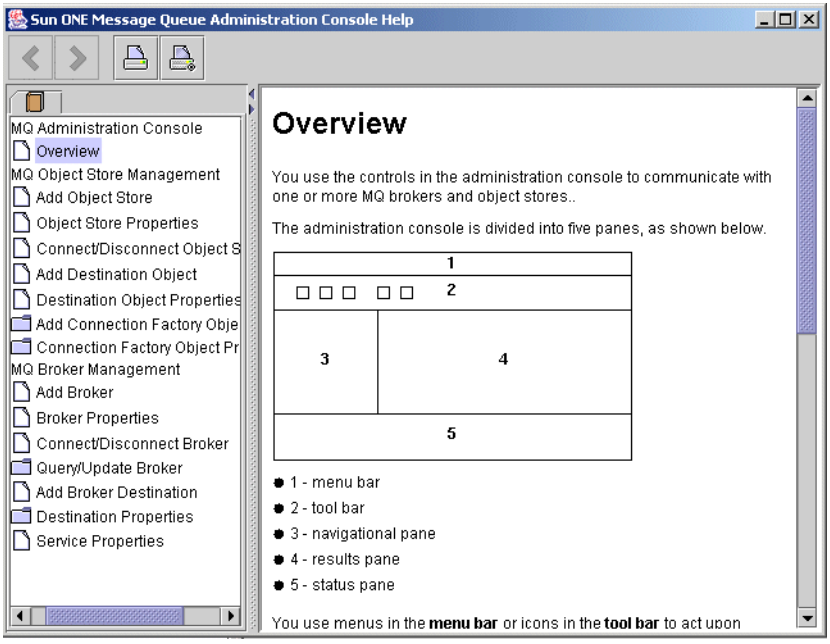
No tutorial can provide complete information, so let's first find out how to get help information for the Administration Console.

Getting Help

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

➤ **To Display Administration Console Help Information**

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



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

Look at the results 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 navigational 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 dialog boxes containing multiple tabs; each file provides help for a simple dialog box or tab.

   Your first Console administration task, "Adding a Broker" on page 106, will be to create a reference to a broker you manage through the Console. Before you start, however, check the online help for information.

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

   Note that the results 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 box. Field names are shown in bold text.

4. Read through the help text.

5. Close the Help window.

Working With Brokers

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

Working with brokers involves the following tasks:

- Start and configure the broker

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

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

- Manage the broker and its services either by using the Administration Console or by using the Command command-line utility (`imqcmd`).
- Create the physical destinations needed by client applications
- Monitor resource use to improve throughput and reliability

The broker supports communication with both application clients and administration clients. It does this by means of different connection services, and you can configure the broker to run any or all of these services. For more information about connection services, see "Connection Services" on page 54.

Starting a Broker

You cannot start a broker using the Administration Console. Start the broker as described in the following procedure (also, see Chapter 5, "Starting and Configuring a Broker").

➤ **To Start a Broker**

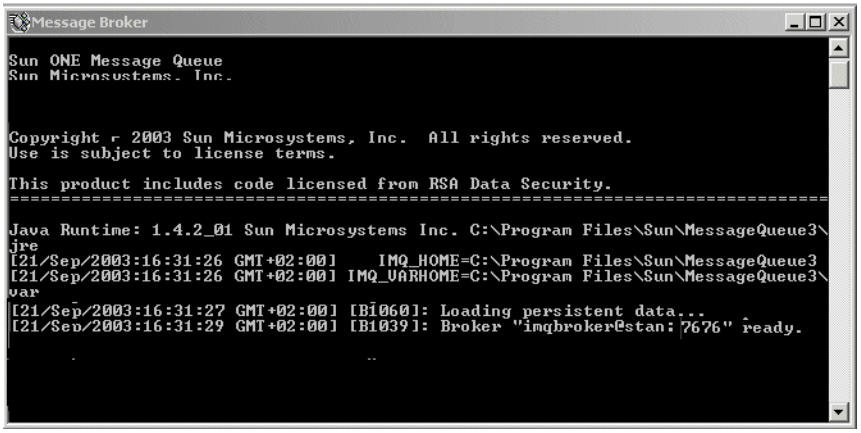
1. Choose Start > Programs >Sun ONE Message Queue 3.5 > Message Broker.

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

   `/usr/bin/imqbrokerd` (on Solaris)

   `/opt/imq/bin/imqbrokerd` (on Linux)

   The command prompt window is displayed and indicates that the broker 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.

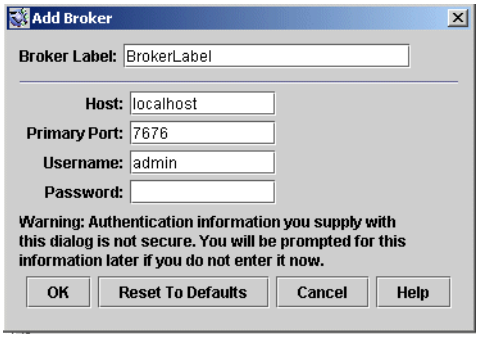
Adding a Broker

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 Brokers in the navigation pane and choose Add Broker.
2. Enter `MyBroker` in the Broker Label field.

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

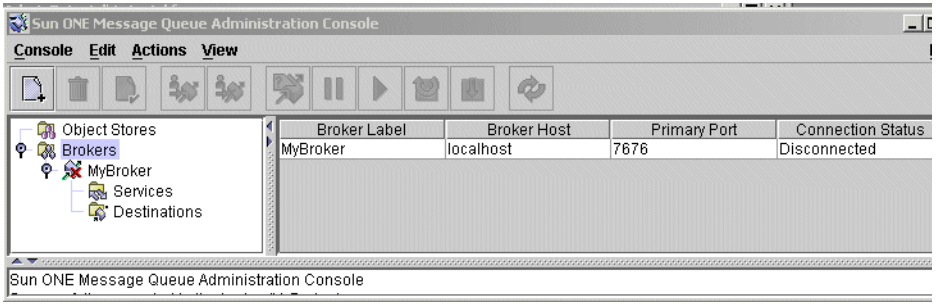


   Note the default host name (`localhost`) and primary port (`7676`) specified in the dialog box. These are the values you will need to specify later, when you configure the connection factory that the client will use to set up connections to this broker.

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

3. Click OK to add the broker.

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



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

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

5. Click Cancel to dismiss the dialog box.

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. By default, the Administration Console can connect to a broker as user `admin` with password `admin`. For improved security, it's a good idea to change the default administrator password (`admin`) before you connect.

➤ **To Change the Administrator Password**

1. Open a command-prompt window or, if one is already opened, bring it forward.

2. Enter a command like the following, substituting your own password for `abracadabra`. The password you specify then replaces the default password of `admin`.

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

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

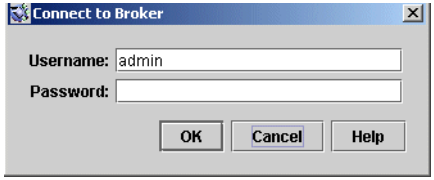
Although clients use a different connection service than administrators, you are also assigned a default user name and password so that you can test MQ without having to do extensive administrative set up. By default, a client can connect to the broker as user `guest` with the password `guest`. You should, however, establish secure user names and passwords for clients as soon as you can. See "Authenticating Users" on page 202 for more information.

Connecting to the Broker

➤ **To Connect to the Broker**

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

   A dialog box 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 108.

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

3. Click OK to connect to the broker.

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

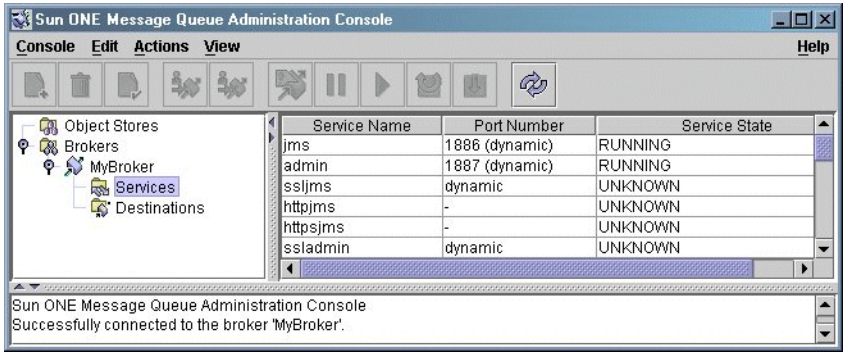
Viewing Connection Services

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

➤ **To View Available Connection Services**

1. Select Services in the navigation pane.

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

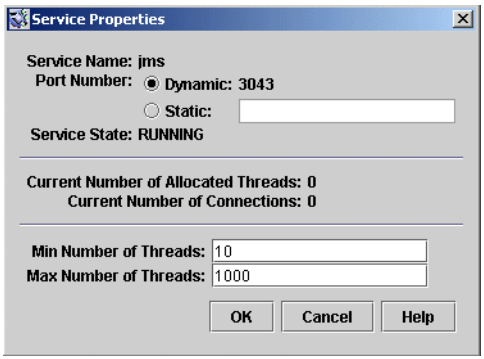


2. Select the jms service by clicking on it in the results pane.
3. Pull down the Actions menu and note the highlighted items.

   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 box, 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 box.
6. Select the admin service in the results pane.
7. Pull down the Actions menu.

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

Adding Physical Destinations to a Broker

By default destination auto-creation is enabled for a broker, which allows it to create physical destinations dynamically. Thus, in a development environment, you do not have to explicitly create destinations in order to test client code.

However, in a production setting, it is advisable to explicitly create physical destinations. This allows you, the administrator, to be fully aware of the destinations that are in use on the broker.

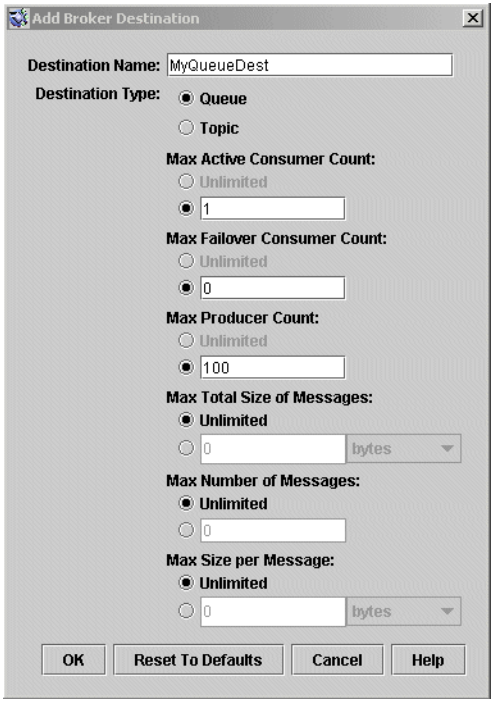
You control whether the broker can add auto-created destinations by setting the `imq.autocreate.topic` or `imq.autocreate.queue` properties. For more information, see "Auto-Created (vs. Admin-Created) Destinations" on page 78.

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

➤ To Add a Queue Destination to a Broker

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

   The following dialog box is displayed:



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

   The destination now appears in the results pane.

Working With Physical Destinations

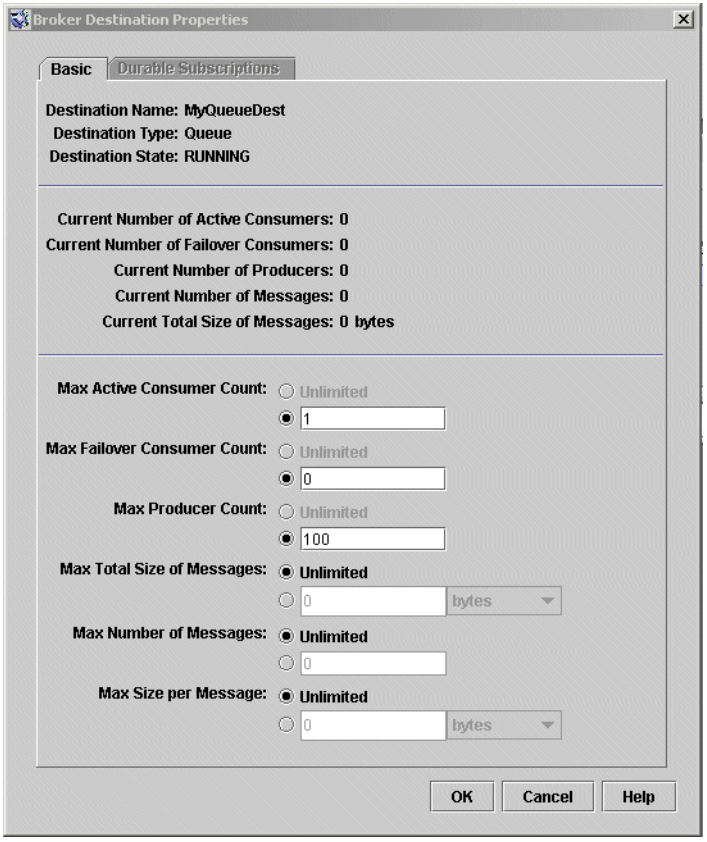
Once you have added a physical destination on the broker, you can do any of the following tasks, as described in the following procedures:

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

➤ **To View the Properties of a Physical Destination**

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

   The following dialog box is displayed:



Note that the dialog box displays current status information about the queue as well as some properties that you can change.

4. Click Cancel to close the dialog box.

➤ To Purge Messages From a Destination

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

   A confirmation dialog box 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.

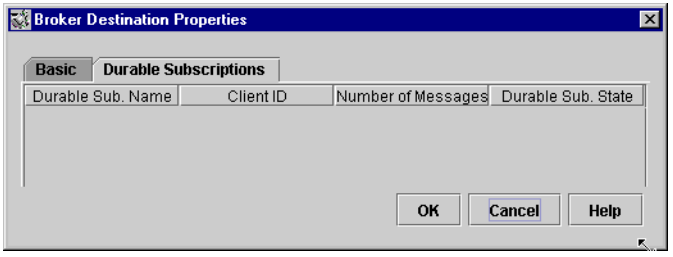
   A confirmation dialog box is displayed.

NOTE Do not delete the MyQueueDest queue destination.

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

Getting Information About Topic Destinations

The broker topic destination properties dialog box includes an additional tab that lists information about durable subscriptions. This tab is disabled for queue destinations.



You can use this dialog box to:

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

Working with Object Stores

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

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

For more information about administered objects, see "MQ Administered Objects" on page 88.

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

Adding an Object Store

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

➤ **To Add a File-system Object Store**

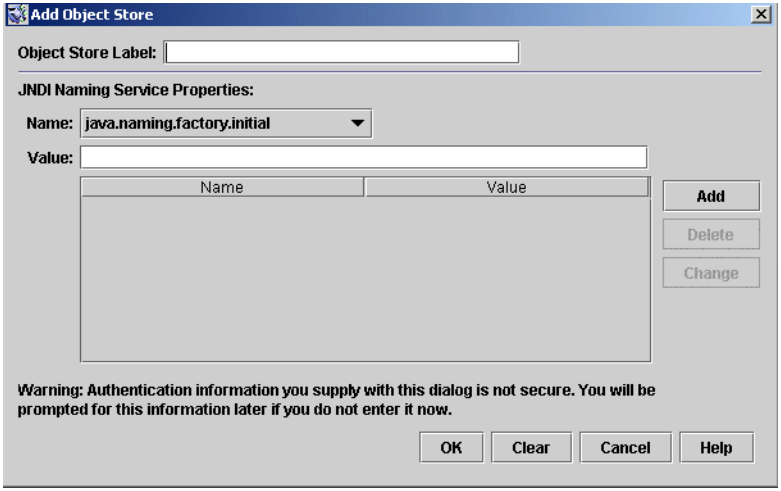
1. If you do not already have a folder named `Temp` on your C drive, create it now.

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

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

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

   The following dialog box 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 drop-down list, select `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 "LDAP Server Object Store" on page 184.

6. Click the Add button.

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

7. From the Name drop-down list, choose `java.naming.provider.url`.

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

8. In the Value field, enter the following

   `file:///C:/Temp`

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

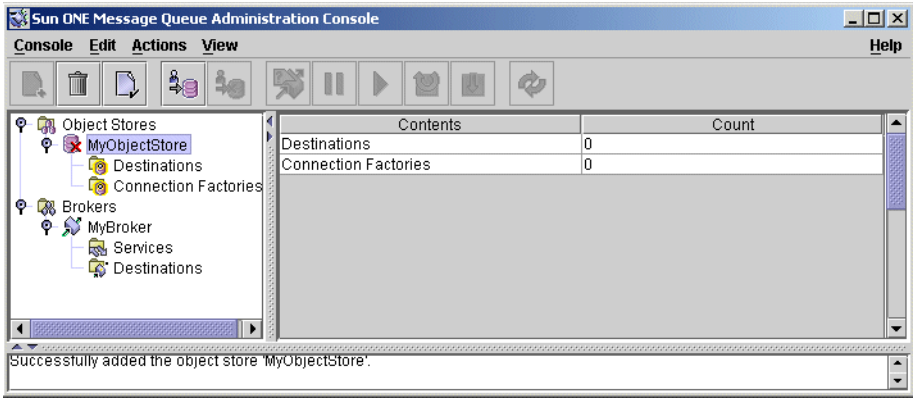
9. Click the Add button.

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

10. Click OK to add the object store.

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

    The Administration Console now looks like this:



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

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

Checking Object Store Properties

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

➤ **To Display the Properties of an Object Store**

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

   A dialog box 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 box.

Connecting to an Object Store

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

➤ **To Connect to an Object Store**

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

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

Adding a Connection Factory Administered Object

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

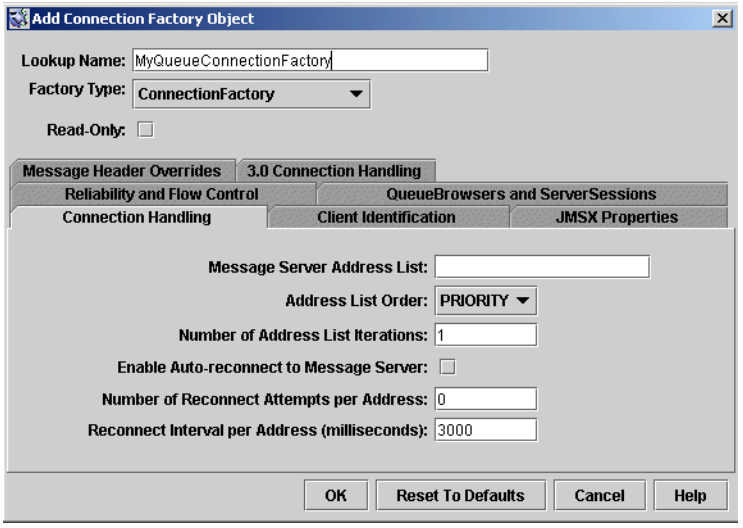
For information on configuring connection factories, see the online help and the MQ *Java Client Developer's Guide*.

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

➤ To Add a Connection Factory to an Object Store

1. If not already connected, connect to MyObjectStore (see "Connecting to an Object Store" on page 119)
2. Right click on the Connection Factories node and choose Add Connection Factory Object.

   The Add Connection Factory Object dialog box is displayed.



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

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

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

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

5. Click the Connection Handling tab.
6. The Message Server Address List field is where you would normally enter the address of the broker to which the client will connect. An example for this field looks like this:

   ```
   mq://localhost:7676/jms
   ```

   You do not need to enter a value since, by default, the connection factory is configured to connect to a broker running on the localhost on port 7676, which is the configuration that the tutorial example expects.
7. Click through the tabs for this dialog box to see the kind of information that you can configure for the connection factory. Use the Help button in the lower right hand corner of the Add Connection Factory Object dialog box to get information about individual tabs. Do not change any of the default values for now.
8. Click OK to create the queue connection factory.
9. Look at the results pane: the lookup name and type of the newly created connection factory are listed.

Adding a Destination Administered Object

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

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

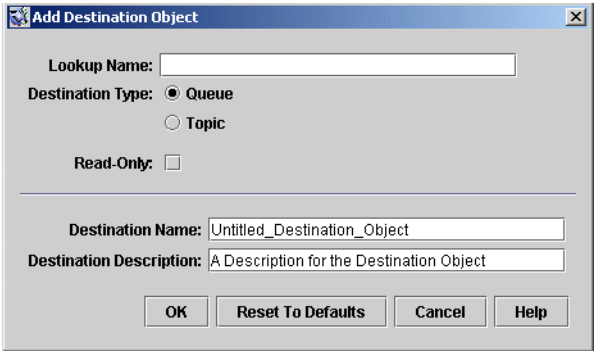
- If you have created a physical destination that is associated with that administered object, the broker delivers the message to that physical destination.
- If you have not created a physical destination and the autocreation of physical destinations is enabled, the broker itself creates the physical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

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

➤ To Add a Destination to an Object Store

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

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



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

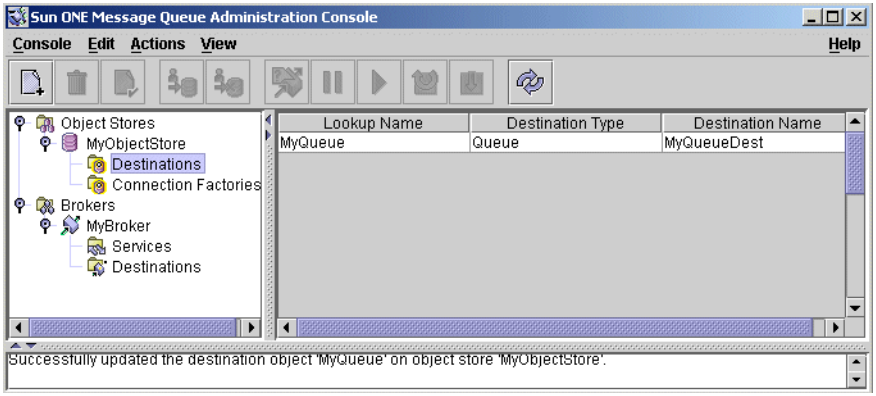
   The lookup name is used to find the object using JNDI lookup calls. In the sample application, the call is the following:

   ```
   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 (see "Adding Physical Destinations to a Broker" on page 111).

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



Administered Object Properties

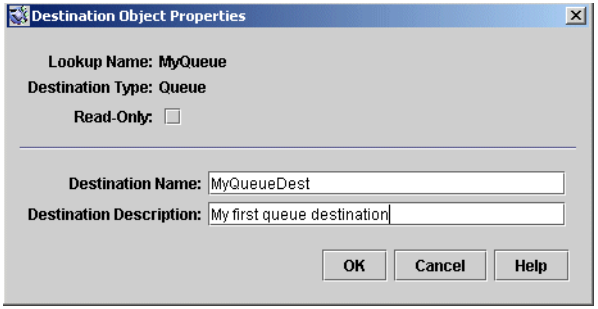
To view or update the properties of an administered object, you need to select Destinations or Connection Factories in the navigation pane, select a specific object in the results pane, and choose Actions > Properties.

➤ **To View or Update the Properties of a Destination Object**

1. Select the Destinations node of MyObjectStore in the navigational pane.
2. Select MyQueue in the results pane.
3. Choose Actions > Properties to view the Destination Object Properties dialog box.

   Note that the only values you can change are 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.

4. Click Cancel to dismiss the dialog box.



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 HelloWorldMessageJNDI is provided for use with this tutorial (for the location, see Step 1, below). It uses the physical destination and administered objects that you created in the foregoing tutorial: a queue physical destination named MyQueueDest, a queue connection factory administered object and queue administered object with JNDI lookup names MyQueueConnectionFactory and MyQueue respectively.

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

➤ **To Run the HelloWorldMessageJNDI Application**

1. Make the directory that includes the `HelloWorldmessageJNDI` application your current directory; for example:

   `cd IMQ_HOME\demo\helloworld\helloworldmessagejndi` (on Windows)

   `cd /usr/demo/imq/helloworld/helloworldmessagejndi` (on Solaris)

   `cd /opt/imq/demo/helloworld/helloworldmessagejndi` (on Linux)

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

2. Set the `CLASSPATH` variable to include the current directory containing the file `HelloWorldMessageJNDI.class` as well as the following jar files that are included in the MQ product: `jms.jar`, `imq.jar`, and `fscontext.jar`. See the MQ *Java Client Developer's Guide* for instructions on setting the `CLASSPATH`.

   The JNDI jar file (jndi.jar) file is bundled with JDK 1.4. If you are using this JDK, you do not have to add `jndi.jar` to your `CLASSPATH` setting. If you are using an earlier version of the JDK, you must include `jndi.jar` in your `CLASSPATH`. See the MQ *Java Client Developer's Guide* for additional information)

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

4. Run the `HelloWorldMessageJNDI` application by executing one of the commands below:

   `java HelloWorldMessageJNDI` (Windows)

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

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

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


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


Creating connection to broker.
Connection to broker created.

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

Chapter 5

Starting and Configuring a Broker

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

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

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

For a description of how to start and use the broker as a Windows service, see "Using a Broker as a Windows Service" on page 331.

Configuration Files

Installed broker configuration file templates, which are used to configure the broker, are located in a directory that depends on operating system, as shown in Appendix A, "Location of MQ Data."

This directory stores the following files:

- **A default configuration file** that is loaded on startup. This file is called `default.properties` and is not editable. You might need to read this file to determine default settings and to find the exact names of properties you want to change.

- **An installation configuration file** that contains any properties specified when MQ is installed. This file is called `install.properties`; it cannot be edited after installation.

Instance Configuration File

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. The instance configuration file is stored in a directory identified by the name of the broker instance (*instanceName*) with which the configuration file is associated (see Appendix A, "Location of MQ Data"):

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

NOTE The …/`instances`/*instanceName* directory (and the instance configuration file) is owned by whoever created the corresponding broker instance. All subsequent start-ups of the broker instance must be by that same user.

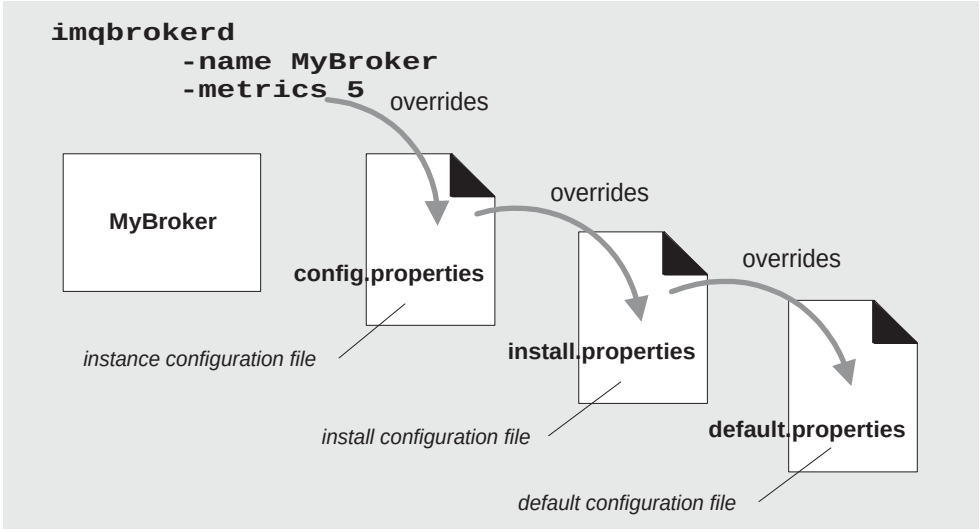
The instance configuration file is maintained by the broker instance. It is modified when you make configuration changes using administration tools. You can also edit an instance configuration file by hand to make configuration changes (see "Editing the Instance Configuration File" on page 129). To do so, you must be the owner of the …/`instances`/*instanceName* directory or log in as root to change privileges on the directory.

If you connect broker instances in a cluster (see "Multi-Broker Clusters (Enterprise Edition)" on page 81) you may also need to use a *cluster configuration file* to specify cluster configuration information. For more information, see "Cluster Configuration Properties" on page 140.

Merging Property Values

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

Figure 5-1 Broker Configuration Files



Property Naming Syntax

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

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

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

```
imq.system.max_count=50000
```

The following entry specifies that a new log file will be created every day (86400 seconds):

```
imq.log.file.rolloversecs=86400
```

Table 5-1 on page 130 lists the broker configuration properties (and their default values) in alphabetical order.

Editing the Instance Configuration File

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

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

- use administration tools. For information about properties you can set using `imqcmd`, see Table 6-4 on page 160.
- edit the `config.properties` file while the broker instance is shut down; then restart the instance. (On Solaris and Linux platforms, only the user that first started the broker instance has permission to edit the `config.properties` file.)

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

Table 5-1 Broker Instance Configuration Properties

Property Name	Type	Default Value	Reference
`imq.accesscontrol.enabled`	boolean	`true`	Table 2-6 on page 69
`imq.accesscontrol.file.filename`	string	`accesscontrol.properties`	Table 2-6 on page 69
`imq.authentication.basic.user_repository`	string	`file`	Table 2-6 on page 69
`imq.authentication.client.response.timeout`	integer (seconds)	`180`	Table 2-6 on page 69
`imq.authentication.type`	string	`digest`	Table 2-6 on page 69
`imq.autocreate.destination.isLocalOnly`	boolean	`false`	Table 2-10 on page 79
`imq.autocreate.destination.limitBehavior`	string	`REJECT_NEWEST`	Table 2-10 on page 79
`imq.autocreate.destination.maxBytesPerMsg`	byte string[1]	`10k`	Table 2-10 on page 79
`imq.autocreate.destination.maxNumMsgs`	integer	`100,000`	Table 2-10 on page 79
`imq.autocreate.destination.maxNumProducers`	integer	`100`	Table 2-10 on page 79
`imq.autocreate.destination.maxTotalMsgBytes`	byte string[1]	`10m`	Table 2-10 on page 79
`imq.autocreate.queue`	boolean	`true`	Table 2-10 on page 79

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

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

Property Name	Type	Default Value	Reference
`imq.autocreate.queue.consumerFlowLimit`	integer	`1000`	Table 2-10 on page 79
`imq.autocreate.queue.localDeliveryPreferred`	boolean	`false`	Table 2-10 on page 79
`imq.autocreate.queue.maxNumActiveConsumers`	integer	`1`	Table 2-10 on page 79
`imq.autocreate.queue.maxNumBackupConsumers`	integer	`0`	Table 2-10 on page 79
`imq.autocreate.topic`	boolean	`true`	Table 2-10 on page 79
`imq.autocreate.topic.consumerFlowLimit`	integer	`1,000`	Table 2-10 on page 79
`imq.cluster.`*property_name*			Table 5-3 on page 141
`imq.hostname`	string	all available IP addresses	Table 2-3 on page 57
`imq.httpjms.http.`*property_name*			Table C-1 on page 309
`imq.httpsjms.https.`*property_name*			Table C-3 on page 321
`imq.keystore.`*property_name*			Table 8-8 on page 220
`imq.log.console.output`	string	`ERROR|WARNING`	Table 2-9 on page 74
`imq.log.console.stream`	string	`ERR`	Table 2-9 on page 74
`imq.log.file.dirpath`	string	See Appendix A, "Location of MQ Data"	Table 2-9 on page 74
`imq.log.file.filename`	string	`log.txt`	Table 2-9 on page 74
`imq.log.file.output`	string	`ALL`	Table 2-9 on page 74
`imq.log.file.rolloverbytes`	integer (bytes)	`-1` `(no rollover)`	Table 2-9 on page 74
`imq.log.file.rolloversecs`	integer (seconds)	`604800`	Table 2-9 on page 74
`imq.log.level`	string	`INFO`	Table 2-9 on page 74
`imq.log.syslog.facility`	string	`LOG_DAEMON`	Table 2-9 on page 74
`imq.log.syslog.identity`	string	`imqbrokerd_${imq.`*instanceName*`}`	Table 2-9 on page 74

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

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

Property Name	Type	Default Value	Reference
imq.log.syslog.logconsole	boolean	false	Table 2-9 on page 74
imq.log.syslog.logpid	boolean	true	Table 2-9 on page 74
imq.log.syslog.output	string	ERROR	Table 2-9 on page 74
imq.log.timezone	string	local time zone	Table 2-9 on page 74
imq.message.expiration.interval	integer (seconds)	60	Table 2-4 on page 62
imq.message.max_size	byte string[1]	70m	Table 2-4 on page 62
imq.metrics.enabled	boolean	true	Table 2-9 on page 74
imq.metrics.interval	integer (seconds)	-1 (never)	Table 2-9 on page 74
imq.metrics.topic.enabled	boolean	true	Table 2-9 on page 74
imq.metrics.topic.interval	integer (seconds)	60	Table 2-9 on page 74
imq.metrics.topic.persist	boolean	false	Table 2-9 on page 74
imq.metrics.topic.timetolive	integer (seconds)	300	Table 2-9 on page 74
imq.passfile.dirpath	string	See Appendix A, "Location of MQ Data"	Table 2-6 on page 69
imq.passfile.enabled	boolean	false	Table 2-6 on page 69
imq.passfile.name	string	passfile	Table 2-6 on page 69
imq.persist.file.destination.message.filepool.limit	integer	100	Table 2-5 on page 66
imq.persist.file.message.cleanup	boolean	false	Table 2-5 on page 66
imq.persist.file.message.filepool.cleanratio	integer	0	Table 2-5 on page 66
imq.persist.file.message.max_record_size	byte string [1]	1m	Table 2-5 on page 66
imq.persist.file.sync.enabled	boolean	false	Table 2-5 on page 66

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

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

Property Name	Type	Default Value	Reference
`imq.persist.jdbc.`*property_name*			Table B-1 on page 298
`imq.persist.store`	string	`file`	Table 2-5 on page 66
`imq.ping.interval`	integer	`120`	Table 2-3 on page 57
`imq.portmapper.backlog`	integer	`50`	Table 2-3 on page 57
`imq.portmapper.hostname`	string	inherited from `imq.hostname`	Table 2-3 on page 57
`imq.portmapper.port`	integer	`7676`	Table 2-3 on page 57

`imq.`*resource_state*`.count`	integer (percent)	`5000 (green)`
		`500 (yellow)`
		`50(orange)`
		`0 (red)`
			Table 2-4 on page 62

`imq.`*resource_state*`.threshold`	integer (percent)	`0 (green)`
		`80 (yellow)`
		`90(orange)`
		`98 (red)`
			Table 2-4 on page 62

`imq.service.activelist`	list	`jms,admin`	Table 2-3 on page 57
`imq.`*service_name*`.accesscontrol.enabled`	boolean	inherits value from system-wide property	Table 2-6 on page 69
`imq.`*service_name*`.accesscontrol.file.filename`	string	inherits value from system-wide property	Table 2-6 on page 69
`imq.`*service_name*`.authentication.type`	string	inherits value from system-wide property	Table 2-6 on page 69
`imq.`*service_name*`.max_threads`	integer	`1000` (jms) `500` (ssljms) `500` (httpjms) `500` (httpsjms) `10` (admin) `10` (ssladmin)	Table 2-3 on page 57
`imq.`*service_name*`.min_threads`	integer	`10` (jms) `10` (ssljms) `10` (httpjms) `10` (httpsjms) `4` (admin) `4` (ssladmin)	Table 2-3 on page 57

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

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

Property Name	Type	Default Value	Reference
`imq.`*service_name*`.`*protocol_type*`.hostname`	string	inherited from `imq.hostname`	Table 2-3 on page 57
`imq.`*service_name*`.`*protocol_type*`.port`	integer	`0` (dynamically allocated)	Table 2-3 on page 57
`imq.`*service_name*`.threadpool_model`	string	`dedicated` (jms) `dedicated` (ssljms) `dedicated` (httpjms) `dedicated` (httpsjms) `dedicated` (admin) `dedicated` (ssladmin)	Table 2-3 on page 57
`imq.shared.connectionMonitor_limit`	integer	`512` (Solaris & Linux) 64 (Windows)	Table 2-3 on page 57
`imq.system.max_count`	integer, 0 (no limit)	`-1`	Table 2-4 on page 62
`imq.system.max_size`	byte string[1], 0 (no limit)	`-1`	Table 2-4 on page 62
`imq.transaction.autorollback`	boolean	`false`	Table 2-4 on page 62
`imq.user_repository.ldap.`*property_name*			Table 8-5 on page 210

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

Starting a Broker

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

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

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

Syntax of the imqbrokerd Command

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

```
imqbrokerd [[ -Dproperty=value]…]
    [ -backup fileName]
    [ -cluster "[broker1] [[,broker2]…]"
    [ -dbuser userName] [ -dbpassword password]
    [ -force]
    [ -h|-help]
    [ -javahome path]
    [ -ldappassword password]
    [ -license licenseName]
    [ -loglevel level]
    [ -metrics interval]
    [ -name instanceName]
    [ -password keypassword] [ -passfile fileName]
    [ -port number]
    [ -remove instance]
    [ -reset data]
    [ -restore fileName]
    [ -shared]
    [ -silent|-s] [ -tty]
    [ -upgrade-store-nobackup]
    [ -version]
    [ -vmargs arg1 [[arg2]…]
```

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

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

Startup Examples

The following examples show the use of the imqbrokerd command. For more details on the imqbrokerd command line options, see Table 5-2 on page 136.

➤ **To Start a Broker Instance That Uses the Default Broker Name and Configuration**

Use the following command:

```
imqbrokerd
```

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

➤ **To Start a Broker Instance With a Trial Enterprise Edition License**

If you have a Platform Edition license, but wish to try out Enterprise Edition features for a period of 90 days, you can enable a trial Enterprise Edition license, using the `-license` command line option and passing "`try`" as the license to use:

```
imqbrokerd -license try
```

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

➤ **To Start a Named Broker Instance With Plugged-in Persistence**

To start a broker named myBroker that uses a plugged-in data store (see Appendix B, "Setting Up Plugged-in Persistence" on page 295) and which requires a username and password, use the following command:

```
imqbrokerd -name myBroker -dbuser myName -dbpassword myPassword
```

Summary of imqbrokerd Options

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

Table 5-2 `imqbrokerd` Options

Option	Properties Affected	Description
`-backup` *fileName*	None affected.	Applies only to broker clusters. Backs up a Master Broker's configuration change record to the specified file. See "Backing up the Configuration Change Record" on page 146.

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

Option	Properties Affected	Description
`-cluster"[`*broker1*`][[,`*broker2*`]...]"` where *broker* is either • *host*[:*port*] • [*host*]:*port*	Sets `imq.cluster.brokerlist` to the list of brokers to which to connect.	Applies only to broker clusters. Connects to all the brokers on the specified hosts and ports. This list is merged with the list in the `imq.cluster.brokerlist` property. If you don't specify a value for *host*, `localhost` is used. If you don't specify a value for *port*, the value 7676 is used. See "Working With Clusters (Enterprise Edition)" on page 140 for more information on how to use this option to connect multiple brokers.
`-dbpassword` *password*	Sets `imq.persist.jdbc.password` to specified password	Specifies the password for a plugged-in JDBC-compliant data store. See Appendix B, "Setting Up Plugged-in Persistence."
`-dbuser` *userName*	Sets `imq.persist.jdbc.user` to specified user name	Specifies the user name for a plugged-in JDBC-compliant database. See Appendix B, "Setting Up Plugged-in Persistence."
`-D`*property=value*	Sets system properties. Overrides corresponding property value in instance configuration file.	Sets the specified property to the specified value. See Table 5-1 on page 130 for broker configuration properties. **Caution:** Be careful to check the spelling and formatting of properties set with the D option. If you pass incorrect values, the system will not warn you, and MQ will not be able to set them.
`-force`	None affected.	Performs action without user confirmation. This option applies only to the `-remove instance` and the `-upgrade-store-nobackup` options, which normally require confirmation.
`-h|-help`	None affected.	Displays help. Nothing else on the command line is executed.
`-javahome` *path*	None affected.	Specifies the path to an alternate Java 2- compatible JDK. The default is to use the bundled runtime.
`-ldappassword` *password*	Sets `imq.user_repository.ldap.password` to specified password	Specifies the password for accessing a LDAP user repository. See "Using an LDAP Server for a User Repository" on page 209.

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

Option	Properties Affected	Description
`-license` [*licenseName*]	None affected.	Specifies the license to load, if different from the default for your MQ product edition. If you don't specify a license name, this lists all licenses installed on the system. Depending on the installed MQ edition, the values for *licenseName* are `pe` (Platform Edition—basic features), `try` (Platform Edition—90-day trial enterprise features), and `unl` (Enterprise Edition). See "Product Editions" on page 32.
`-loglevel` *level*	Sets `imq.broker.log.level` to the specified level.	Specifies the logging level as being one of `NONE`, `ERROR`, `WARNING`, or `INFO`. The default value is `INFO`. For more information, see "Logger" on page 71.
`-metrics` *interval*	Sets `imq.metrics.interval` to the specified number of seconds.	Specifies that broker metrics be written to the Logger at an interval specified in seconds.
`-name` *instanceName*	Sets `imq.instancename` to the specified name.	Specifies the instance name of this broker and uses the corresponding instance configuration file. If you do not specify a broker name, the name of the instance is set to `imqbroker`. **Note**: If you run more than one instance of a broker on the same host, each must have a unique name.
`-passfile` *fileName*	Sets `imq.passfile.enabled` to `true`. Sets `jmq.passfile.dirpath` to the path that contains the file. Sets `imq.passfile.name` to the name of the file.	Specifies the name of the file from which to read the passwords for the SSL keystore, LDAP user repository, or JDBC-compliant database. For more information, see "Using a Passfile" on page 224.
`-password` *keypassword*	Sets `imq.keystore.password` to the specified password.	Specifies the password for the SSL certificate keystore. For more information, see "Security Manager" on page 66.
`-port` *number*	Sets `imq.portmapper.port` to the specified number.	Specifies the broker's Port Mapper port number. By default, this is set to 7676. To run two instances of a broker on the same server, each broker's Port Mapper must have a different port number. MQ clients connect to the broker instance using this port number.
`-remove instance`	None affected.	Causes the broker instance to be removed: deletes the instance configuration file, log files, persistent store, and other files and directories associated with the instance. Requires user confirmation unless `-force` option is also specified.

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

Option	Properties Affected	Description
`-reset store|messages|durables|props`	None affected.	Resets the data store (or a subset of the data store) or the configuration properties of a broker instance, depending on the argument given. Resetting the data store clears out all persistent data, including persistent messages, durable subscriptions, and transaction information. This allows you to start the broker instance with a clean slate. You can also clear only all persistent messages or only all durable subscriptions. (If you do not want the persistent store to be reset on subsequent restarts, then re-start the broker instance without using the `-reset` option.) For more information, see "Persistence Manager" on page 63. Resetting the broker's properties, replaces the existing instance configuration file (`config.properties`) with an empty file: all properties assume default values.
`-restore` *fileName*	None affected.	Applies only to broker clusters. Replaces the Master Broker's configuration change record with the specified backup file. This file must have been previously created using the `-backup` option. See "Restoring the Configuration Change Record" on page 146.
`-shared`	Sets `imq.jms.threadpool_model` to `shared`.	Specifies that the jms connection service be implemented using the shared threadpool model, in which threads are shared among connections to increase the number of connections supported by a broker instance. For more information, see "Connection Services" on page 54.
`-silent|-s`	Sets `imq.log.console.output` to `NONE`.	Turns off logging to the console.
`-tty`	Sets `imq.log.console.output` to `ALL`	Specifies that all messages be displayed to the console. By default only `WARNING` and `ERROR` level messages are displayed.

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

Option	Properties Affected	Description
`-upgrade-store-nobackup`	None affected	Specifies that an earlier, incompatible version MQ data store is automatically removed when migrating to MQ 3.5 format. If you don't use this option, you must manually delete the earlier data store. This applies to both built-in (flat-file) persistence and plugged-in (JDBC-compliant) persistence. Migration of the earlier data store to a MQ 3.5 data store takes place the first time you start a MQ 3.5 broker instance on an earlier version data store.
`-version`	None affected.	Displays the version number of the installed product.
`-vmargs` *arg1* `[[`*arg2*`]…]`	None affected	Specifies arguments to pass to the Java VM. Separate arguments with spaces. If you want to pass more than one argument or if an argument contains a space, use enclosing quotation marks. For example: `imqbrokerd -tty -vmargs "-Xmx128m -Xincgc"`

Working With Clusters (Enterprise Edition)

This section describes the properties you use to configure multi-broker clusters, describes two methods of connecting brokers, and explains how to manage clusters. For an introduction to clusters, see "Multi-Broker Clusters (Enterprise Edition)" on page 81.

When working with clusters, make sure that you synchronize clocks among the hosts of all brokers in a cluster (see "System Clock Settings" on page 337).

Cluster Configuration Properties

When you connect brokers into a cluster, all the connected brokers must specify as set of cluster configuration properties. These properties describe the participation of the brokers in a cluster. Table 5-3 summarizes the cluster-related configuration properties. Properties marked with an asterisk (*) must have the same value for all brokers in a cluster.

Table 5-3 Cluster Configuration Properties

Property Name	Description
`imq.cluster.brokerlist*`	Specifies all the brokers in a cluster. Consists of a comma-separated list of *host:port* entries, where *host* is the host name of each broker and *port* is its Port Mapper port number. For example: `host1:3000, host2:8000, ctrhost`
`imq.cluster.masterbroker*`	Specifies which broker in a cluster (if any) is the Master Broker that keeps track of state changes. Property consists of *host:port* where *host* is the host name of the Master Broker and *port* is its Port Mapper port number. Set this property for production environments. For example, `ctrhost:7676`

`imq.cluster.url*`	Specifies the location of a cluster configuration file. Used in cases where brokers reference one central cluster configuration file rather than being individually configured. 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*.
	
	For example:
	`http://webserver/imq/cluster.properties`
	`file:/net/mfsserver/imq/cluster.properties`

`imq.cluster.port`	For *each* broker within a cluster, can be used to specify the port number for the `cluster` connection service. The `cluster` connection service is used for internal communication between brokers in a cluster. Default: `0` (port is dynamically allocated)
`imq.cluster.hostname`	For *each* broker within a cluster, can be used to specify the host (hostname or IP address) to which the `cluster` connection service binds if there is more than one host available (for example, if there is more than one network interface card in a computer). The `cluster` connection service is used for internal communication between brokers in a cluster. Default: inherits the value of `imq.hostname` (see Table 2-3 on page 57)
`imq.cluster.transport*`	Specifies the network transport used by the `cluster` connection service for internal communication between brokers in a cluster. For secure, encrypted message delivery between brokers, set this property to `ssl` for *all* brokers in a cluster. Default: `tcp`

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

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

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

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

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

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

The following code sample shows the contents of a cluster configuration file. Both `host1` and `ctrlhost` are running on the default port. These properties specify that `host1`, `host2`, and `ctrlhost` are connected in a cluster and that `ctrlhost` is the Master Broker.

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

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

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

Connecting Brokers

This section describes how to connect brokers into a cluster and how to configure the cluster for secure, encrypted message delivery between brokers in the cluster.

Connection Methods

There are two general methods of connecting brokers into a cluster: connecting with or without a cluster configuration file.

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 Master Broker is started up. 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: Connecting Without a Cluster Configuration File

➤ **To Connect Brokers into a Cluster**

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

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

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

Method 2: Connecting With 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. If you use this method, each broker in the cluster must set the value of the `imq.cluster.url` property to point to the cluster configuration file.

Secure Inter-Broker Connections

In situations where you want secure, encrypted message delivery between the brokers in a cluster, you have to configure the `cluster` connection service to use an SSL-based transport protocol, as follows.

➤ **To Configure Secure Connections Within a Cluster**

1. For each broker in the cluster, set up SSL-based connection services.

   See the instructions in "Setting Up an SSL-based Service Over TCP/IP" on page 218.

2. Set the `imq.cluster.transport` cluster configuration property to `ssl`.

   If you are not using a cluster configuration file, you need to set this property for each broker in the cluster.

Managing Brokers in a Cluster

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

Adding Brokers to a Cluster

➤ **To Add a New Broker to an Existing Cluster**

- If you are using a cluster configuration file, then

  a. Add the new broker to the `imq.cluster.brokerlist` property in the cluster configuration file.

  b. Issue the following command to every broker in the cluster.

     ```
     imqcmd reload cls
     ```

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

  c. Start the new broker specifying the `imq.cluster.url` property on the command line using the `-D` option.

     This points the broker to the cluster configuration file.

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

Restarting a Broker in a Cluster

If a broker in a cluster crashed or was shut down for some reason, you need to restart it as a member of the cluster.

➤ **To Restart a Broker That is Already a Member of an Existing Cluster**

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

Removing a Broker from a Cluster

➤ **To Remove a Broker From an Existing Cluster**

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

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

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

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

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

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

  a. Remove mention of the broker from the configuration file.

  b. Change or remove the `imq.cluster.url` property for the broker that is being removed so that it no longer uses the common properties.

  c. Use the `imqcmd reload cls` command to force all the brokers to reload their cluster configuration and thereby reconfigure the cluster.

Managing 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. The state includes information about durable subscriptions and administrator-created physical destinations. All brokers consult the Master Broker during startup (which, in turn, consults its configuration change record) in order to synchronize information about these persistent objects. Consequently, the failure of the Master Broker would make such synchronization impossible. As a result, if the Master Broker fails, you cannot create or delete physical destinations or durable subscriptions.

Because of the important information it contains, it is important that you back up the Master Broker's configuration change record regularly and restore it in case of failure.

The following sections explain how to back up and restore the configuration change record.

Backing up the Configuration Change Record

➤ **To Back Up the Configuration Change Record**

Use the `-backup` option of the `imqbrokerd` command. For example,

```
imqbrokerd -backup mybackuplog
```

It is important you do this in a timely manner. Restoring a very old backup can result in loss of information: any changes in physical destinations or durable subscriptions since the backup was last done will be lost.

Restoring the Configuration Change Record

➤ **To Restore the Master Broker in Case of Failure**

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

   ```
   imqbrokerd -restore mybackuplog
   ```

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

4. Restart all the brokers.

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

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

Logging

This section describes the default logging configuration for the broker and explains how you can change that configuration to redirect log information to alternate output channels and change log file rollover criteria. For an introduction to logging, see "Logger" on page 71. For information on using logging to report broker metrics, see "Monitoring Tools" on page 244.

Default Logging Configuration

When you start the broker, it is automatically configured to save log output to a set of rolling log files located in a directory identified by the name of the broker instance (*instanceName*) with which the log files are associated (see Appendix A, "Location of MQ Data"):

…/`instances/`*instanceName*`/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 `imq.log.file.dirpath` to the desired path.
- To change the root name of the log files from `log` to something else, set the `imq.log.file.filename` property.

The broker supports three log categories: `ERROR`, `WARNING`, `INFO` (see Table 2-7 on page 72). Setting a logging level gathers messages for that level and all higher levels. The default log level is `INFO`. This means that `ERROR`, `WARNING`, and `INFO` messages are all logged by default.

Log Message Format

Logged messages consist of a timestamp (see Table 2-9 on page 74 to change the timestamp time zone), message code, and the message itself. The volume of information varies with the log level you have set. The following is an example of an `INFO` message.

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

Changing the Logger Configuration

All Logger properties are described in Table 2-9 on page 74.

➤ **To Change the Logger Configuration for a Broker**

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

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

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

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

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

imqbrokerd Options	Description
`-metrics` *interval*	Specifies the interval (in seconds) at which metrics information is written to the Logger.
`-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

Changing the Output Channel

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

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

- To have *all* log categories (for a given level) output displayed on the screen, use the `-tty` option to the `imqbrokerd` command.
- To prevent log output from being displayed on the screen, use the `-silent` option to the `imqbrokerd` command.

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

  ```
  imq.log.file.output=ERROR
  ```

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

  ```
  imq.log.console.output=INFO
  ```

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

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

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

Changing Log File Rollover Criteria

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

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

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

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

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

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

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
- managing connection services

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

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

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

Command Utility

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

Syntax of the imqcmd Command

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

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

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

```
imqcmd restart bkr -v
```

imqcmd Subcommands

The Command utility (`imqcmd`) includes the subcommands listed in Table 6-1: The subcommands are described in more detail in the task-oriented sections of this chapter.

Table 6-1 `imqcmd` Subcommands

Subcommand and Argument	Description
`commit txn`	Commits a transaction.
`compact dst`	Compacts the built-in file-based data store for one or more destinations.
`create dst`	Creates a destination.
`destroy dst`	Destroys a destination.
`destroy dur`	Destroys a durable subscription.
`list cxn`	Lists connections for a broker.
`list dst`	Lists destinations on a broker.
`list dur`	Lists durable subscriptions to a topic.

Table 6-1 imqcmd Subcommands *(Continued)*

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

Summary of imqcmd Options

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

Table 6-2 `imqcmd` Options

Option	Description
`-b` *hostName:port*	Specifies the name of the broker's host and its port number. The default value is `localhost:7676`. To specify port only: `-b :7878` To specify name only: `-b somehost`
`-c` *clientID*	Specifies the ID of the durable subscriber to a topic. See "Managing Durable Subscriptions" on page 179.
`-d` *destinationName*	Specifies the name of the topic. Used with the `list dur` and `destroy dur` subcommands. See "Managing Durable Subscriptions" on page 179.
`-f`	Performs action without user confirmation.
`-h`	Displays usage help. Nothing else on the command line is executed.
`-H`	Displays usage help, attribute list, and examples. Nothing else on the command line is executed.
`-int` *interval*	Specifies the interval, in seconds, at which the `metrics bkr`, `metrics dst`, and `metrics svc` subcommands display metrics output.
`-javahome` *path*	Specifies an alternate Java 2 compatible runtime to use (default is to use the runtime on the system or the runtime bundled with MQ).
`-m` *metricType*	Specifies the type of metric information to display. Use this option with the `metrics dst`, `metrics svc`, or `metrics bkr` subcommand. The value of *metricType* depends on whether the metrics are generated for a destination, a service, or a broker.
`-msp` *numSamples*	Specifies the number of metric samples the `metrics bkr`, `metrics dst`, and `metrics svc` subcommands display in their metrics output.
`-n` *argumentName*	Specifies the name of the subcommand argument. Depending on the subcommand, this might be the name of a service, a physical destination, a durable subscription, a connection ID, or a transaction ID.
`-o` *attribute=value*	Specifies the value of an attribute. Depending on the subcommand argument, this might be the attribute of a broker (see "Managing a Broker" on page 157), service (see "Managing Connection Services" on page 162), or destination (see "Managing Destinations" on page 168).

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

Option	Description
`-p` *password*	Specifies your (the administrator's) password. If you omit this value, you will be prompted for it.
`-pst` *pauseType*	Specifies whether producers, consumers, or both are paused when pausing a destination. See "Managing Destinations" on page 168.
`-rtm` *timeout*	Specifies the initial (retry) timeout period (in seconds) of an `imqcmd` subcommand. The timeout is the length of time the `imqcmd` subcommand will wait after making a request to the broker. Each subsequent retry of the subcommand will use a timeout value that is a multiple of the initial timeout period. Default: `10`
`-rtr` *numRetries*	Specifies the number of retries attempted after an `imqcmd` subcommand first times out. Default: `5`
`-s`	Silent mode. No output will be displayed.
`-secure`	Specifies a secure administration connection to the broker using the `ssladmin` connection service (see "Step 4. Configuring and Running SSL-based Clients" on page 222).
`-svn` *serviceName*	Specifies the service for which connections are listed. See "Getting Connection Information" on page 167.
`-t` *destType*	Specifies the type of a destination: `t` (topic) or `q` (queue). See "Managing Destinations" on page 168.
`-tmp`	Displays temporary destinations. See Table 6-9 on page 168.
`-u` *userName*	Specifies your (the administrator's) name. If you omit this value, you will be prompted for it.
`-v`	Displays version information. Nothing else on the command line is executed.

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

NOTE To be able to use the `-secure` option, you must first set up and enable the `ssladmin` service on the target broker instance, as described in "Setting Up an SSL-based Service Over TCP/IP" on page 218.

Using imqcmd Commands

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

- Start the broker using the `imqbrokerd` command.

  See "Starting a Broker" on page 134. You can use the Command utility only to administer brokers that are already running; you cannot use it to start a broker.

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

- Specify the proper administrator user name and password. If you do not do this, you will be prompted for it. Either way, be aware that every operation you perform using `imqcmd` will be authenticated against a user repository. For more information, see "Authenticating Users" on page 202.

  When you install MQ, a default flat-file user repository is installed. 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 instance without doing any additional work. For example, if you are just testing MQ, you can run the `imqcmd` utility using the 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 administrative users (see Chapter 8, "Managing Security"). In particular, you need to make entries in the MQ user repository (see "Using a Flat-File User Repository" on page 202). You also have the option of using an LDAP directory server for your user repository (see "Using an LDAP Server for a User Repository" on page 209).

Example imqcmd Usage

The following examples illustrate the use of the `imqcmd` command:

- To list the properties of the broker running on `localhost` at port `7676`:

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

- To list the properties of the broker running on `myserver` at port `1564`; the user's name is `alladin`, the user's password is `abracadabra`.

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

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

- To list the properties of the broker running on `localhost` at port `7676`, with the initial timeout for the command set to 20 seconds and the number of retries (after timeout) set to 7.

  ```
  imqcmd query bkr -u admin -p admin -rtm 20 -rtr 7
  ```

Managing a Broker

The Command utility includes subcommands that you can use to perform the following broker management tasks:

- Displaying Broker Information
- Updating Broker Properties
- Displaying Broker Metrics
- Controlling the Broker's State

To manage connection services for a broker, see "Managing Connection Services" on page 162. To manage broker destinations, see "Managing Destinations" on page 168

Table 6-3 lists the `imqcmd` subcommands used to manage brokers. If no host name or port is specified, the default (`localhost:7676`)is assumed.

Table 6-3 `imqcmd` Subcommands Used to Manage a Broker

Subcommand Syntax	Description

`metrics bkr [-b` *hostName:port*`]`
`[-m` *metricType*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
	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` Displays metrics on messages and packets flowing into and out of the broker. (default metric type)
	`rts` Displays metrics on rate of flow of messages and packets into and out of the broker (per second).
	`cxn` Displays connections, virtual memory heap, and threads.
	Use the `-int` option to specify the interval (in seconds) at which to display the metrics. The default is 5 seconds.
	Use the `-msp` option to specify the number of samples displayed in the output. The default is an unlimited number (infinite).

`pause bkr [-b` *hostName:port*`]`	Pauses the default broker or a broker at the specified host and port. See "Pausing and Resuming a Broker" on page 161.
`query bkr -b` *hostName:port*	Lists the current settings of properties of the default broker or a broker at the specified host and port. Also shows the list of running brokers (in a multi-broker cluster) that are connected to the specified broker.
`reload cls`	Applies only to broker clusters. Forces all the brokers in a cluster to reload the `imq.cluster.brokerlist` property and update cluster information. See "Adding Brokers to a Cluster" on page 144 for more information.
`restart bkr [-b` *hostName:port*`]`	Shuts down and restart the default broker or a broker at the specified host and port. Note that this command restarts the broker using the options specified when the broker was first started. If you want different options to be in effect, you must shutdown the broker and then start it again, specifying the options you want.
`resume bkr [-b` *hostName:port*`]`	Resumes the default broker or a broker at the specified host and port.
`shutdown bkr [-b` *hostName:port*`]`	Shuts down the default broker or a broker at the specified host and port.

Table 6-3 `imqcmd` Subcommands Used to Manage a Broker *(Continued)*

Subcommand Syntax	Description

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

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

Displaying Broker Information

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

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

This command produces output like the following:

```
Version                                              3.5
Instance Name                                        imqbroker
Primary Port                                         7676

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

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

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

Cluster Broker List (active)
Cluster Broker List (configured)
Cluster Master Broker
Cluster URL

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

Updating Broker Properties

You can use the `update bkr` subcommand to update any of the broker properties listed in Table 6-4. Note that updates to the broker are automatically written to the broker's instance configuration file.

Table 6-4 Broker Properties Updated by `imqcmd`

Property	Reference
`imq.autocreate.queue`	Table 2-10 on page 79
`imq.autocreate.topic`	Table 2-10 on page 79
`imq.autocreate.queue.maxNumActiveConsumers`	Table 2-10 on page 79
`imq.autocreate.queue.maxNumBackupConsumers`	Table 2-10 on page 79
`imq.cluster.url`	Table 5-3 on page 141.
`imq.log.level`	Table 2-9 on page 74
`imq.log.file.rolloversecs`	Table 2-9 on page 74
`imq.log.file.rolloverbytes`	Table 2-9 on page 74
`imq.system.max_count`	Table 2-4 on page 62
`imq.system.max_size`	Table 2-4 on page 62
`imq.message.max_size`	Table 2-4 on page 62
`imq.portmapper.port`	Table 2-3 on page 57

For example, the following command turns off the auto-creation of queue destinations:

```
imqcmd update bkr -o "imq.autocreate.queue=false"
                  -u admin -p admin
```

Controlling the Broker's State

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

Pausing and Resuming a Broker

- **Pausing the broker**. Pausing a broker suspends the broker's connection service threads, which causes the broker to stop listening on the connection ports. As a result, the broker will no longer be able to accept new connections, receive messages, dispatch messages.

  However, pausing a broker does not suspend the admin connection service, letting you perform 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.

  Pausing a broker also does not suspend the cluster connection service. However message delivery within a cluster depend on the delivery functions performed by the different brokers in the cluster.

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

  ```
  imqcmd pause bkr -b myhost:1588 -u admin -p admin
  ```

  (You can also pause individual connection services—see "Pausing and Resuming a Connection Service" on page 166—as well as individual destinations—see "Pausing and Resuming Destinations" on page 175)

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

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

Shutting Down and Restarting a Broker

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

  ```
  imqcmd shutdown bkr -b ctrlsrv:1572 -u admin -p admin
  ```

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

  ```
  imqcmd restart bkr -u admin -p admin
  ```

Displaying Broker Metrics

To display metrics information about a broker, use the `metrics bkr` subcommand. For example, to get the rate of message flow into and out of the broker at ten second intervals:

```
imqcmd metrics bkr -m rts -int 10 -u admin -p admin
```

This command produces output like the following:

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

For a more detailed description of the use of `imqcmd` to report broker metrics, see "Monitoring Tools" on page 244.

Managing Connection Services

The Command utility includes subcommands that allow you to perform the following connection service management tasks:

- Listing Connection Services
- Displaying Connection Service Information
- Updating Connection Service Properties
- Displaying Connection Service Metrics
- Pausing and Resuming a Connection Service

For an overview of MQ connection services, see "Connection Services" on page 54.

Table 6-5 lists the `imqcmd` subcommands used to manage connection services. If no host name or port is specified, the default (`localhost:7676`)is assumed.

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

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

`metrics svc -n` *serviceName*
`[-b` *hostName:port*`]`
`[-m` *metricType*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
	Displays 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` Displays metrics on messages and packets flowing into and out of the broker by way of the specified service. (default metric type)
	`rts` Displays metrics on rate of flow of messages and packets into and out of the broker (per second) by way of the specified service.
	`cxn` Displays connections, virtual memory heap, and threads.
	Use the `-int` option to specify the interval (in seconds) at which to display the metrics. The default is 5 seconds.
	Use the `-msp` option to specify the number of samples displayed in the output. The default is an unlimited number (infinite).

`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.
`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.
`resume svc -n` *serviceName* `[-b` *hostName:port*`]`	Resumes the specified service running on the default broker or on a broker at the specified host and port.
`update svc -n` *serviceName* `[-b` *hostName:port*`]` `-o` *attribute=value* `[-o` *attribute=value1*`]`...	Updates the specified attribute of the specified service running on the default broker or on a broker at the specified host and port. For a description of service attributes, see Table 6-7 on page 165.

A broker supports connections from both application clients and administration clients. The connection services currently available from an MQ 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 (application clients) or ADMIN (administration clients)—and an underlying transport layer.

Table 6-6 Connection Services Supported by a Broker

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

Listing Connection Services

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

```
imqcmd list svc [-b hostName:portNumber] -u admin -p admin
```

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

```
imqcmd list svc -b MyServer:6565 -u admin -p admin
```

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

```
imqcmd list svc -u admin -p admin
```

The command will output information like the following:

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

```
jms             41843 (dynamic)    RUNNING
ssladmin        dynamic            UNKNOWN
ssljms          dynamic            UNKNOWN
```

Displaying Connection Service Information

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

```
imqcmd query svc -n jms -u admin -p admin
```

This command produces output like the following:

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

Current Number of Allocated Threads   0
Current Number of Connections         0

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

Updating Connection Service Properties

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

Table 6-7 Connection Service Properties Updated by `imqcmd`

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

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

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

Displaying Connection Service Metrics

To display metrics information about a single service, use the `metrics` subcommand. For example, to get cumulative totals for messages and packets handled by the jms connection service:

```
imqcmd metrics svc -n jms -m ttl -u admin -p admin
```

This command produces output like the following:

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

For a more detailed description of the use of `imqcmd` to report connection service metrics, see “Monitoring Tools” on page 244.

Pausing and Resuming a Connection Service

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

```
imqcmd pause svc -n serviceName -u admin -p admin
```

Pausing a service has the following effects:

- The broker stops accepting new client connections on the paused service. If an MQ client attempts to open a new connection, it will get an exception.
- All the existing connections on the paused service are kept alive, but the broker suspends all message processing on such connections until the service is resumed. (For example, if a client attempts to send a message, the `send()` method will block until the service is resumed.)

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

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

```
imqcmd resume svc -n serviceName -u admin -p admin
```

Getting Connection Information

The Command utility includes subcommands that allow you to list and get information about connections.

Table 6-8 lists the `imqcmd` subcommands that apply to connections. If no host name or port is specified, they are assumed to be `localhost`, `7676`.

Table 6-8 `imqcmd` Subcommands Used to Manage Connection Services

Subcommand Syntax	Description
`list cxn [-svn` *serviceName*`]` `[-b` *hostName:port*`]`	Lists all connections of the specified service name on the default broker or on a broker at the specified host and port. If the service name is not specified, all connections are listed.
`query cxn -n` *connectionID* `[-b` *hostName:port*`]`	Displays information about the specified connection on the default broker or on a broker at the specified host and port.

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

```
imqcmd query cxn -n 421085509902214374 -u admin -p admin
```

This command produces output like the following:

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

Managing Destinations

All MQ messages are routed to their consumer clients by way of queue and topic destinations created on a particular broker.

The Command utility includes subcommands that allow you to perform the following destination management tasks:

- Creating Destinations
- Listing Destinations
- Displaying Destination Information
- Updating Destination Attributes
- Displaying Destination Metrics
- Pausing and Resuming Destinations
- Purging Destinations
- Destroying Destinations
- Compacting Destinations

For an introduction to destinations, see "Physical Destinations" on page 76.

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

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

Subcommand Syntax	Description
`compact dst [-t` *destType* `-n` *destName*`]`	Compacts the built-in file-based data store for the destination of the specified type and name. If no destination type and name are specified, then all destinations are compacted. Destinations must be paused before they can be compacted.
`create dst -t` *destType* `-n` *destName* `[-o` *attribute*`=`*value*`]` `[-o` *attribute*`=`*value*`1]...`	Creates a destination of the specified type, with the specified name, and the specified attributes. Destination names must contain only alphanumeric characters (no spaces) and can begin with an alphabetic character or the characters "_" and "$". They cannot begin with the character string "`mq.`"
`destroy dst -t` *destType* `-n` *destName*	Destroys the destination of the specified type and name.

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

Subcommand Syntax	Description
`list dst [-t` *destType*`] [-tmp]`	Lists all destinations of the specified type, with option of listing temporary destinations as well (see "Temporary Destinations" on page 81). The type argument can have two values: *destType* = `q` (queue) *destType* = `t` (topic) If the type is not specified, all destinations of all types are listed.

`metrics dst -t` *destType*
`-n` *destName*
`[-m` *metricType*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
	Displays metrics information for the destination of the specified type and name.
	
	Use the `-m` option to specify the type of metric to display:
	
	`ttl` Displays metrics on messages and packets flowing into and out of the destination and residing in memory. (default metric type))
	
	`rts` Displays metrics on rate of flow of messages and packets into and out of the destination (per second) and other rate information.
	
	`con` Displays consumer-related metrics.
	
	`dsk` Displays disk usage metrics.
	
	Use the `-int` option to specify the interval (in seconds) at which to display the metrics. The default is 5 seconds.
	
	Use the `-msp` option to specify the number of samples displayed in the output. The default is an unlimited number (infinite).

`pause dst [-t` *destType*
`-n` *destName*`]`
`[-pst` *pauseType*`]`
	Pauses the delivery of messages to consumers (`-pst CONSUMERS`), or from producers (`-pst PRODUCERS`), or both (`-pst ALL`), for the destination of the specified type and name. If no destination type and name are specified, then all destinations are paused. The default is `ALL`.

`purge dst -t` *destType* `-n` *destName*	Purges messages at the destination of the specified type and name.
`query dst -t` *destType* `-n` *destName*	Lists information about the destination of the specified type and name.
`resume dst [-t` *destType* `-n` *destName*`]`	Resumes the delivery of messages for the paused destination of the specified type and name. If no destination type and name are specified, then all destinations are resumed.

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

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

Creating Destinations

When creating a destination, you must specify the following:

- The destination type: topic or queue
- The destination name: must contain only alphanumeric characters (no spaces) and can begin with an alphabetic character or the characters "_" and "$". The name cannot begin with the character string "mq."
- Any non-default values for the destination's attributes

Many of the destination attributes are used to manage broker memory resources and message flow. For example, you can specify the maximum number of producers allowed for a destination or the maximum number (or size) of messages allowed in a destination. These limits are similar to those that can be set on a broker-wide basis using broker configuration properties (see "Managing Memory Resources and Message Flow" on page 61). You can also specify how the broker responds when these limits are reached.

There are also destination attributes that apply only to queue destinations. These are used to specify the number of active and backup consumers used in load-balanced delivery of messages to multiple consumers (see "Queue Destinations" on page 76).

Table 6-10 describes the attributes that apply for each type of destination. You can set the attribute values when you create or update a destination. For auto-created destinations you set default property values in the broker's instance configuration file (see "Configuration Files" on page 127).

Table 6-10 Destination Attributes

Destination Type	Attribute	Default Value	Description
Queue & Topic	`maxNumMsgs`[1]	`-1` (unlimited)	Specifies maximum number of unconsumed messages allowed in the destination.
Queue & Topic	`maxTotalMsgBytes`[1]	`-1` (unlimited)	Specifies the maximum total amount of memory (in bytes) allowed for unconsumed messages in the destination.
Queue & Topic	`limitBehavior`	`REJECT_NEWEST`	Specifies how the broker responds when a memory-limit threshold is reached. Values are: `FLOW_CONTROL` — slows down producers `REMOVE_OLDEST` — throws out oldest messages `REMOVE_LOW_PRIORITY` — throws out lowest priority messages according to age of the messages (producing client receives no notification of message deletion) `REJECT_NEWEST` — rejects the newest messages (producing client gets exception for rejection of persistent messages, but no notification for rejection of non-persistent messages)
Queue & Topic	`maxBytesPerMsg`	`-1` (unlimited)	Specifies maximum size (in bytes) of any single message allowed in the destination (producing client gets exception for rejection of persistent messages, but no notification for rejection of non-persistent messages.
Queue & Topic	`maxNumProducers`[1]	`-1` (unlimited)	Specifies maximum number of producers allowed for the destination. When this limit is reached, no new producers can be created.
Queue only	`maxNumActiveConsumers`	`1`	Specifies the maximum number of consumers that can be active in load-balanced delivery from a queue destination. A value of `-1` means an unlimited number. (Platform Edition limits this value to 2.)
Queue only	`maxNumBackupConsumers`	`0`	Specifies the maximum number of backup consumers that can take the place of active consumers, if any fail during load-balanced delivery from a queue destination. A value of `-1` means an unlimited number.

1. In a cluster environment, this property applies to each instance of the destination in the cluster, rather than collectively to all instances in the cluster.

Table 6-10 Destination Attributes *(Continued)*

Destination Type	Attribute	Default Value	Description
Queue & Topic	`consumerFlowLimit`	Topics: `1000` Queues: `1000`	Specifies the maximum number of messages that will be delivered to a consumer in a single batch. In load-balanced queue delivery, this is the initial number of queued messages routed to active consumers before load-balancing commences (see "Queue Delivery to Multiple Consumers" on page 76). This limit can be overridden by a lower value specified for the destination's consumers on their respective connections (see information on Connection Factory attributes in the MQ *Java Client Developer's Guide*). A value of `-1` means an unlimited number.
Queue only	`localDeliveryPreferred`	`false`	Applies only to load-balanced queue delivery in broker clusters. Specifies that messages be delivered to remote consumers only if there are no consumers on the local broker. Requires that the destination not be restricted to local-only delivery (`isLocalOnly = false`).
Queue & Topic	`isLocalOnly`	`false`	Applies only to broker clusters. Specifies that a destination is not replicated on other brokers, and is therefore limited to delivering messages only to local consumers (consumers connected to the broker on which the destination is created). This attribute cannot be updated once the destination has been created.

1. In a cluster environment, this property applies to each instance of the destination in the cluster, rather than collectively to all instances in the cluster.

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

  ```
  imqcmd create dst -n myQueue -t q -o "maxNumActiveConsumers=5"
  ```

  Note that a destination name must contain only alphanumeric characters (no spaces) and can begin with an alphabetic character or the characters "_" and "$". It cannot begin with the character string "mq," which is reserved for metrics topic destinations (see Table 2-8 on page 73.

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

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

Listing Destinations

You can get information about a destination's current attribute values, about the number of producers or consumers associated with a destination, and about messaging metrics, such as the number and size of messages in the destination.

To find a destination about which you want to get information, you can first list all destinations on a particular broker using the `list dst` subcommand. For example, to get a list of all destinations on the broker running on `myHost` at port 4545, enter the following command:

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

The `list dst` subcommand can optionally specify the type of destination to list or optionally include temporary destinations (using the `-tmp` option). Temporary destinations are created by clients, normally for the purpose of receiving replies to messages sent to other clients (see "Temporary Destinations" on page 81).

Displaying Destination Information

To get information about a destination's current attribute values, use the `query dst` subcommand, such as in the following command:

```
imqcmd query dst -t q -n XQueue -u admin -p admin
```

This command produces output like the following:

```
-----------------------------------
Destination Name    Destination Type
-----------------------------------
XQueue              Queue

On the broker specified by:

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

Destination Name                     XQueue
Destination Type                     Queue
Destination State                    RUNNING
Created Administratively             true

Current Number of Messages           0
Current Total Message Bytes          0
Current Number of Producers          0
Current Number of Active Consumers   0
```

```
Current Number of Backup Consumers    0

Max Number of Messages                unlimited (-1)
Max Total Message Bytes               unlimited (-1)
Max Bytes per Message                 unlimited (-1)
Max Number of Producers               100
Max Number of Active Consumers        1
Max Number of Backup Consumers        0

Limit Behavior                        REJECT_NEWEST
Consumer Flow Limit                   100
Is Local Destination                  false
Local Delivery is Preferred           false
```

The output also shows the number of producers and consumers associated with the destination. For queue destinations, this would include both active and backup consumers.

You can use the `update dst` subcommand to change the values of one or more attributes (see "Updating Destination Attributes" on page 174).

Updating Destination Attributes

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

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

See Table 6-10 on page 171 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 `isLocalOnly` attribute.

Displaying Destination Metrics

To get message metrics information about a destination, use the `metrics dst` subcommand, such as in the following command:

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

This command produces output like the following:

```
---------------------------------------------------------------------------
  Msgs      Msg Bytes         Msg Count         Total Msg Bytes (k)     Largest
In   Out    In     Out    Current  Peak  Avg  Current  Peak     Avg    Msg (k)
---------------------------------------------------------------------------
200  200  147200  147200     0     200    0      0      143      71        0
300  200  220800  147200    100    200   10     71      143      64        0
300  300  220800  220800     0     200    0      0      143      59        0
```

For a more detailed description of the use of `imqcmd` to report destination metrics, see "Monitoring Tools" on page 244.

Pausing and Resuming Destinations

You can pause a destination to control the delivery of messages from producers to the destination, or from the destination to consumers, or both. In particular, you can pause the flow of messages into a destination to help prevent destinations from being overwhelmed with messages when production of messages is much faster than consumption.

To pause the delivery of messages to or from a destination, use the `pause dst` subcommand, as in the following command:

```
imqcmd pause dst -n myQueue -t q -pst PRODUCERS -u admin -p admin
```

```
imqcmd pause dst -n myTopic -t t -pst CONSUMERS -u admin -p admin
```

In the case where you have paused a destination and want to resume delivery, enter the following command:

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

In a multi-broker cluster, instances of the destination reside on each broker in the cluster. You must pause each of these destinations individually.

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, use the `purge dst` subcommand, as in the following commands:

```
imqcmd purge dst -n myQueue -t q -u admin -p admin
```

```
imqcmd purge dst -n myTopic -t t -u admin -p admin
```

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

```
imqbrokerd -reset messages -u admin -p admin
```

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

In a multi-broker cluster, instances of the destination reside on each broker in the cluster. You must purge each of these destinations individually.

Destroying Destinations

To destroy a destination, use the `destroy dst` subcommand, as in the following command:

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

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

Compacting Destinations

If you are using the built-in file-based data store (as opposed to a plugged-in JDBC-compliant data store) as the persistent store for messages, you can monitor disk utilization and compact the disk when necessary.

The file-based message store is structured so that messages are stored in directories according to the destinations in which they are being held. In each destination's directory, most messages are stored in one file consisting of variable-sized records, the variable-sized record file. (To alleviate fragmentation, messages whose size exceeds a configurable threshold will be stored in their own individual files.) As messages of varying sizes are persisted and then removed from the variable-sized record file, holes may develop in the file where free records are not being re-used.

To manage unused free records, the Command utility includes subcommands for monitoring disk utilization per destination and for reclaiming free disk space when utilization drops.

Monitoring a Destination's Disk Utilization

To monitor a destination's disk utilization, use the following `imqcmd` subcommand:

```
imqcmd metrics dst -t q -n myQueue -m dsk -u admin -p admin
```

This command produces output like the following:

```
--------------------------------------
Reserved   Used      Utilization Ratio
--------------------------------------
806400     804096    99
1793024    1793024   100
2544640    2518272   98
```

The columns in the subcommand output have the following meaning:

Table 6-11 Destination disk Utilization Metrics

Metric	Description
Reserved	Disk space in bytes used by all records, including records that hold active messages and free records waiting to be reused
Used	Disk space in bytes used by records that hold active messages
Utilization Ratio	Quotient of used disk space divided by reserved disk space. The higher the ratio, the more the disk space is being used to hold active messages.

Reclaiming Unused Destination Disk Space

The disk utilization pattern depends on the characteristics of the messaging application that uses a particular destination. Depending on the relative flow of messages into and out of a destination, and the relative size of messages, the reserved disk space might grow over time.

If the message producing rate is greater than the message consuming rate, then free records should generally be reused and the utilization ratio should be on the high side. However, if the message producing rate is similar to or smaller than the message consuming rate, you can expect that the utilization ratio will be low.

In general, you want the reserved disk space to stabilize and the utilization to remain high. As a rule of thumb, if the system reaches a steady state in which the amount of reserved disk space stays pretty much constant and the utilization rate is high (above 75%), there is no need to reclaim the unused disk space. If the system reaches a steady state and the utilization rate is low (below 50%), you can compact the disk to reclaim the disk space occupied by free records.

If the reserved disk space continues to increase over time, you should reconfigure the destination's memory management by setting destination memory limit properties and limit behaviors (see Table 6-10 on page 171).

➤ **To Reclaim Unused Destination Disk Space**

1. Pause the destination.

   ```
   imqcmd pause dst -t q -n myQueue -u admin -p admin
   ```

2. Compact the disk.

   ```
   imqcmd compact dst -t q -n myQueue -u admin -p admin
   ```

3. Resume the destination.

   ```
   imqcmd resume dst -t q -n myQueue -u admin -p admin
   ```

If destination type and name are not specified, then these operations are performed for *all* destinations.

Managing Durable Subscriptions

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

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

Table 6-12 `imqcmd` Subcommands Used to Manage Durable Subscriptions

Subcommand	Description
`list dur -d` *destName*	Lists all durable subscriptions for the specified destination.
`destroy dur -n` *subscrName* `-c` *client_id*	Destroys the specified durable subscription with the specified Client Identifier (see "Client Identifiers" on page 44).
`purge dur -n` *subscrName* `-c` *client_id*	Purges all messages for the specified durable subscription with the specified Client Identifier (see "Client Identifiers" on page 44).

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

```
imqcmd list dur -d SPQuotes
```

For each durable subscription to a topic, the `list dur` 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:

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

You can use the information returned from the `list dur` subcommand to identify a durable subscription you might want to destroy or for which you want to purge messages. Use the name of the subscription and the client ID to identify the subscription. For example:

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

Managing Transactions

All transactions initiated by client applications are tracked by the broker. These can be simple MQ transactions or distributed transactions managed by an XA resource manager (see "Local Transactions" on page 46). All transaction have an MQ transaction ID—a 64 bit number that uniquely identifies a transaction on the broker. Distributed transactions also have a distributed transaction ID (XID) assigned by the distributed transaction manager—up to 128 bytes long. MQ maintains the association of an MQ transaction ID with an XID.

For distributed transactions, in cases of failure, it is possible that transactions could be left in a `PREPARED` state without ever being committed. Hence, as an administrator you might need to monitor and then roll back or commit transactions left in a prepared state.

Table 6-13 provides a summary of the `imqcmd` transactions subcommands. Remember to specify the host name and port of the broker if this is not the default (`localhost:7676`) broker.

Table 6-13 `imqcmd` Subcommands Used to Manage Transactions

Subcommand	Description
`list txn`	Lists all transactions, being tracked by the broker.
`query txn -n` *transaction_id*	Lists information about the specified transaction.
`commit txn -n` *transaction_id*	Commits the specified transaction.
`rollback txn -n` *transaction_id*	Rolls back the specified transaction.

For example, the following command lists all transactions in a broker.

```
imqcmd list txn
```

For each transaction, the `list` subcommand returns the transaction ID, state, user name, number of messages or acknowledgements, and creation time. For example:

```
-------------------------------------------------------------
Transaction ID  State     User name   # Msgs/   Creation time
                                      # Acks
-------------------------------------------------------------

64248349708800  PREPARED  guest       4/0       1/30/02 10:08:31 AM
64248371287808  PREPARED  guest       0/4       1/30/02 10:09:55 AM
```

The command shows all transactions in the broker, both local and distributed. You can only commit or roll back transactions in the `PREPARED` state. You should only do so if you know that the transaction has been left in this state by a failure and is not in the process of being committed by the distributed transaction manager.

For example, if the broker's auto-rollback property is set to false (see Table 2-4 on page 62), then you have to manually commit or roll back transactions found in a `PREPARED` state at broker startup.

The `list` subcommand also shows the number of messages that were produced in the transaction and the number of messages that were acknowledged in the transaction (`#Msgs/#Acks`). These messages will not be delivered and the acknowledgements will not be processed until the transaction is committed.

The `query` subcommand lets you see the same information plus a number of additional values: the Client ID, connection identification, and distributed transaction ID (XID). For example,

```
imqcmd query txn -n 64248349708800
```

produces output like the following:

```
Client ID
Connection                guest@192.18.116.219:62209->jms:62195
Creation time             1/30/02 10:08:31 AM
Number of acknowledgements 0
Number of messages        4
State                     PREPARED
Transaction ID            64248349708800
User name                 guest
XID
6469706F6C7369646577696E6465723130313234313431313030373230
```

The `commit` and `rollback` subcommands can be used to commit or roll back a distributed transaction. As mentioned previously, only a transaction in the `PREPARED` state can be committed or rolled back. For example:

```
imqcmd commit txn -n 64248349708800
```

It is also possible to configure the broker to automatically roll back transactions in the `PREPARED` state at broker startup. See the `imq.transaction.autorollback` property in Table 2-4 on page 62 for more information.

Chapter 7

Managing Administered Objects

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

For an overview of administered objects, see"MQ Administered Objects" on page 88.

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

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

This chapter explains how you use the Object Manager utility (`imqobjmgr`) to perform these tasks. Because these tasks involve an understanding of the attributes of both the object store you are using and of the administered objects you are creating, this chapter provides background on these two topics before describing how to use `imqobjmgr` to manage administered objects.

For information using the Administration Console, see Chapter 4, "Administration Console Tutorial."

About Object Stores

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

LDAP Server Object Store

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

MQ administration tools can manage object stores on LDAP servers. However, you might first need to configure the LDAP server to store java objects and perform JNDI lookups, as prescribed in the documentation for the LDAP server.

In using an LDAP server as your object store, you need to specify the attributes shown in Table 7-1. These attributes fall into the following categories:

- **Initial Context**: This attribute is fixed for an LDAP server object store.
- **Location**: Specifies the URL and directory path for storing your administered objects, as set up in the LDAP server. In particular you must check that the specified path exists.
- **Security Information**: Depends on the LDAP provider. You should 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.

Table 7-1 LDAP Object Store Attributes

Attribute	Description
`java.naming.factory.initial`	The initial context for a JNDI lookup on an LDAP server `com.sun.jndi.ldap.LdapCtxFactory`
`java.naming.provider.url`	LDAP server URL and directory path information. For example: `ldap://mydomain.com:389/ou=mqobjs, o=myapp` where administered objects are stored in the `/myapp/mqobjs` directory

Table 7-1 LDAP Object Store Attributes *(Continued)*

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

File-system Object Store

MQ also supports a file-system object store implementation. While the file-system object store is not fully tested and is therefore not recommended for production systems, it has the advantage of being very easy to use in development environments. Rather than setting up an LDAP server, all you have to do is create a directory on your local file system.

However a file-system store cannot be used as a centralized object store for clients deployed across multiple computer nodes unless these clients have access to the directory where the object store resides. In addition, any user with access to that directory can use MQ administration tools to create and manage administered objects.

In using a file-system object store, you need to specify the attributes shown in Table 7-2. These attributes fall into the following categories:

- **Initial Context**: The value of this attribute is fixed for a file system object store.
- **Location**: The value of this attribute specifies the directory path for storing your administered objects. The directory must exist and have the proper access permissions for the user of MQ administration tools as well as the users of the client applications that will access the store.

Table 7-2 File-system Object Store Attributes

Attribute	Description
`java.naming.factory.initial`	The initial context for a JNDI lookup on a file system object store: `com.sun.jndi.fscontext.RefFSContextFactory`
`java.naming.provider.url`	Directory path information. For example: `file:///C:/myapp/mqobjs`

Administered Objects

For an overview of administered objects, see "MQ Administered Objects" on page 88.

MQ administered objects are of two basic kinds: connection factories and destinations. *Connection factory* administered objects are used by client applications to create a connection to a broker. *Destination* administered objects are used by client applications to identify the destination to which a producer is sending messages or from which a consumer is retrieving messages. (A special *SOAP endpoint* administered object is used for SOAP messaging—see the MQ *Java Client Developer's Guide* for more information.)

Depending on the message delivery model (point-to-point or publish/subscribe), connection factories and destinations of a specific type can be used. In point-to-point programming, for example, a queue connection factory and a queue destination can be used. Similarly, in publish and subscribe programming, a topic connection factory and a topic destination can be used. Non-specific connection factory and destination administered object types are also available, as are connection factory types that support distributed transactions (see Table 1-1 on page 44 for all the supported types).

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

Connection Factory Administered Object Attributes

Connection factory (and XA connection factory) administered objects have the attributes listed in Table 7-3. The attribute you are primarily concerned with is `imqAddressList`, which you use to specify the broker to which the client will establish a connection. The section, "Adding a Connection Factory" on page 195, explains how you specify a attributes when you add a connection factory administered object to your object store.

For more descriptions of connection factory attributes and information on how they are used, see the MQ *Java Client Developer's Guide* and the JavaDoc API documentation for the following MQ class: `com.sun.messaging.ConnectionConfiguration`.

Table 7-3 Connection Factory Administered Object Attributes

Attribute/property name	Type	Default Value
`imqAckOnAcknowledge`	String	No value
`imqAckOnProduce`	String	No value
`imqAckTimeout`	String	`0 millisecs`
`imqAddressList`	String	No value
`imqAddressListIterations`	Integer	`1`
`imqAddressListBehavior`	String	`PRIORITY`
`imqBrokerHostName` (MQ 3.0)	String	`localhost`
`imqBrokerHostPort` (MQ 3.0)	Integer	`7676`
`imqBrokerServicePort` (MQ 3.0)	Integer	`0`
`imqConfiguredClientID`	String	No value
`imqConnectionFlowCount`	Integer	`100`
`imqConnectionFlowLimit`	Integer	`1000`
`imqConnectionFlowLimitEnabled`	Boolean	`false`
`imqConnectionType` (MQ 3.0)	String	`TCP`

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

Attribute/property name	Type	Default Value
`imqConnectionURL` (MQ 3.0)	String	`http://localhost/imq/tunnel`
`imqConsumerFlowLimit`	Integer	`1000`
`imqConsumerFlowThreshold`	Integer	`50`
`imqDefaultPassword`	String	`guest`
`imqDefaultUsername`	String	`guest`
`imqDisableSetClientID`	Boolean	`false`
`imqJMSDeliveryMode`	Integer	`2` (persistent)
`imqJMSExpiration`	Long	`0` (does not expire)
`imqJMSPriority`	Integer	`4` (normal)
`imqLoadMaxToServerSession`	Boolean	`true`
`imqOverrideJMSDeliveryMode`	Boolean	`false`
`imqOverrideJMSExpiration`	Boolean	`false`
`imqOverrideJMSHeadersTo TemporaryDestinations`	Boolean	`false`
`imqOverrideJMSPriority`	Boolean	`false`
`imqQueueBrowserMaxMessages PerRetrieve`	Integer	`1000`
`imqQueueBrowserRetrieveTimeout`	Long	`60,000` (milliseconds)
`imqReconnectAttempts`	Integer	`0`
`imqReconnectEnabled`	Boolean	`false`
`imqReconnectInterval`	Long	`3000` (milliseconds)
`imqSetJMSXAppID`	Boolean	`false`
`imqSetJMSXConsumerTXID`	Boolean	`false`
`imqSetJMSXProducerTXID`	Boolean	`false`
`imqSetJMSXRcvTimestamp`	Boolean	`false`
`imqSetJMSXUserID`	Boolean	`false`
`imqSSLIsHostTrusted` (MQ 3.0)	Boolean	`true`

Destination Administered Object Attributes

The destination administered object that identifies a physical topic or queue destination has the attributes listed in Table 7-4. The section, "Adding a Topic or Queue" on page 196, explains how you specify these attributes when you add a destination administered object to your object store.

The attribute you are primarily concerned with is `imqDestinationName`. This is the name you assign to the physical destination that corresponds to the topic or queue administered object. You can also provide a description of the destination that will help you distinguish it from others that you might create to support many applications.

For more information, see the JavaDoc API documentation for the MQ class `com.sun.messaging.DestinationConfiguration`.

Table 7-4 Destination Administered Object Attributes

Attribute/property name	Type	Default
`imqDestinationDescription`	String	`A Description for the destination Object`
`imqDestinationName`	String[1]	`Untitled_Destination_Object`

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

Object Manager Utility (imqobjmgr)

The Object Manager utility allows you to create and manage MQ administered objects. This section describes the basic `imqobjmgr` command syntax, provides a listing of subcommands, and summarizes `imqobjmgr` command options. Subsequent sections explain how you use the `imqobjmgr` subcommands to accomplish specific tasks.

Syntax of the imqobjmgr Command

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

```
imqobjmgr subcommand [options]
imqobjmgr -h|H
imqobjmgr -v
```

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

```
imqobjmgr list -v
```

imqobjmgr Subcommands

The Object Manager utility (`imqobjmgr`) includes the subcommands listed in Table 7-5:

Table 7-5 `imqobjmgr` Subcommands

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

Summary of imqobjmgr Command Options

Table 7-6 lists the options to the `imqobjmgr` command. For a discussion of their use, see the task-based sections that follow.

Table 7-6 `imqobjmgr` Options

Option	Description
`-f`	Performs action without user confirmation.
`-h`	Displays usage help. Nothing else on the command line is executed.
`-H`	Displays usage help, attribute list, and examples. Nothing else on the command line is executed.
`-i` *fileName*	Specifies the name of an command file containing all or part of the subcommand clause, specifying object type, lookup name, object attributes, object store attributes, or other options. Typically used for repetitive information, such as object store attributes.

Table 7-6 `imqobjmgr` Options *(Continued)*

Option	Description
`-j` *attribute=value*	Specifies attributes necessary to identify and access a JNDI object store. See "LDAP Server Object Store" on page 184 and "File-system Object Store" on page 185.
`-javahome` *path*	Specifies an alternate Java 2 compatible runtime to use (default is to use the runtime on the system or the runtime bundled with MQ).
`-l` *lookupName*	Specifies the JNDI lookup name of an administered object. This name must be unique in the object store's context.
`-o` *attribute=value*	Specifies attributes of an administered object. See "Connection Factory Administered Object Attributes" on page 187 and "Destination Administered Object Attributes" on page 189
`-pre`	Preview mode. Indicates what will be done without performing the command.
`-r` *read-only_state*	Specifies whether an administered object is a read-only object. A value of `true` indicates the administered object is a read-only object. Clients cannot modify the attributes of read-only administered objects. The read-only state is set to `false` by default.
`-s`	Silent mode. No output will be displayed.
`-t` *objectType*	Specifies the type of an MQ administered object: q = queue t = topic cf = connection factory qf = queue connection factory tf = topic connection factory xcf = XA connection factory (distributed transactions) xqf = XA queue connection factory (distributed transactions) xtf = XA topic connection factory (distributed transactions) e = SOAP endpoint[1]
`-v`	Displays version information. Nothing else on the command line is executed.

1. This administered object type is used to support SOAP messages (see the MQ *Java Client Developer's Guide*).

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

Required Information

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

- **The administered object type**:

  The allowed types are shown in Table 7-6.

- **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.

- **Administered object attributes** (needed especially for the `add` and `update` subcommands):

  - For destinations: The name of the physical destination on the broker. This is the name that was specified with the `-n` option to the `imqcmd create dst` subcommand. If you do not specify the name, the default name of `Untitled_Destination_Object` will be used.

  - For connection factories: The most commonly used attribute is the address list (`imqAddressList`) specifying the message server addresses (one or more) to which the client will attempt to connect. If you do not specify this information, the local host and default port number (7676) are used, meaning the client will attempt a connection to a broker on port `7676` of the local host. The section "Adding a Connection Factory" on page 195 explains how you specify object attributes.

    For additional attributes, see "Connection Factory Administered Object Attributes" on page 187.

- **Object store attributes:**

  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 "LDAP Server Object Store" on page 184 and "File-system Object Store" on page 185.

Using Command Files

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

Using a command file with the Object Manager utility (`imqobjmgr`) is especially useful to specify object store attributes, which are likely to be the same across multiple invocations of `imqobjmgr` and which normally require a lot of typing. Using an command 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 an `imqobjmgr` command file is as follows (the version property reflects the version of the command file and not of the MQ product—it is not a command line option—and its value must be set to `2.0`):

```
version=2.0
cmdtype=[ add | delete | list | query | update ]
obj.type=[ q | t | qf | tf | cf | xqf | xtf | xcf | e ]
obj.lookupName=lookup name
obj.attrs.objAttrName1=value1
obj.attrs.objAttrName2=value2
obj.attrs.objAttrNameN=valueN
…
objstore.attrs.objStoreAttrName1=value1
objstore.attrs.objStoreAttrName2=value2
objstore.attrs.objStoreAttrNameN=valueN
…
```

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

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

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

```
version=2.0
cmdtype=add
obj.type=qf
obj.lookupName=cn=myQCF
obj.attrs.imqAddressList=mq://foo:777/jms
objstore.attrs.java.naming.factory.initial=\
             com.sun.jndi.ldap.LdapCtxFactory
objstore.attrs.java.naming.provider.url=\
             ldap://mydomain.com:389/o=imq
  objstore.attrs.java.naming.security.principal=\
             uid=fooUser, ou=People, o=imq
  objstore.attrs.java.naming.security.credentials=fooPasswd
  objstore.attrs.java.naming.security.authentication=simple
```

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

```
imqobjmgr -i MyCmdFile
```

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

```
imqobjmgr add
       -t qf
       -l "cn=myQCF"
       -o "imqAddressList=mq://foo:777/jms"
       -i MyCmdFile
```

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

```
version=2.0
objstore.attrs.java.naming.factory.initial=\
             com.sun.jndi.ldap.LdapCtxFactory
objstore.attrs.java.naming.provider.url=\
```

```
                ldap://mydomain.com:389/o=imq
      objstore.attrs.java.naming.security.principal=\
                uid=fooUser, ou=People, o=imq
      objstore.attrs.java.naming.security.credentials=fooPasswd
      objstore.attrs.java.naming.security.authentication=simple
```

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

```
IMQ_HOME/demo/imqobjmgr
```

Adding and Deleting Administered Objects

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

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

Adding a Connection Factory

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

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

```
imqobjmgr add
    -t qf
    -l "cn=myQCF"
    -o "imqAddressList=mq://myHost:7272/jms"
    -j "java.naming.factoryinitial=
                com.sun.jndi.ldap.LdapCtxFactory"
    -j "java.naming.provider.url=ldap://mydomain.com:389/o=imq"
    -j "java.naming.security.principal=
```

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

The preceding command creates an administered object whose lookup name is `cn=myQCF` and which connects to a broker running on `myHost` and listens on port 7272. The administered object is stored in an LDAP server. You can accomplish the same thing by specifying an command file as an argument to the `imqobjmgr` command. For more information, see "Using Command Files" on page 193.

NOTE **Naming Conventions:** 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 (`cn=myQCF`). You specify the lookup name with the `-l` option. You do not have to use the `cn` prefix if you are using a file-system object store, however do not use lookup names that have a "/" in them. See Table 7-7.

Table 7-7 Naming Convention Examples

Object Store Type	Good Name	Ban Name
LDAP server	cn=myQCF	myQCF
file system	myTopic	myObjects/myTopic

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 a good practice to first create the physical destinations before adding the corresponding administered objects to the object store. Use the Command utility (`imqcmd`) to create the physical destinations on the broker that are identified by destination administered objects in the object store. For information about creating physical destinations, see "Getting Connection Information" on page 167.

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

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

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

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

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

```
version=2.0
cmdtype=add
obj.type=t
obj.lookupName=cn=myTopic
obj.attrs.imqDestinationName=TestTopic
objstore.attrs.java.naming.factory.initial=
            com.sun.jndi.fscontext.RefFSContextFactory
objstore.attrs.java.naming.provider.url=
            file:///home/foo/imq_admin_objects
  objstore.attrs.java.naming.security.principal=
            uid=fooUser, ou=People, o=imq
  objstore.attrs.java.naming.security.credentials=fooPasswd
  objstore.attrs.java.naming.security.authentication=simple
```

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

```
imqobjmgr -i MyCmdFile
```

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

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

Deleting Administered Objects

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

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

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

Getting Information

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

Listing Administered Objects

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

The following command lists all objects.

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

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

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

Information About a Single Object

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

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

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

Updating Administered Objects

You use the `update` command to modify the attributes of administered objects. You must specify the lookup name and location of the object. You use the -o option to modify attribute values.

This command changes the attributes of an administered object that represents a topic connection factory:

```
imqobjmgr update
 -t tf
 -l "cn=MyTCF"
 -o imqReconnectAttempts=3
 -j "java.naming.factory.initial=
            com.sun.jndi.ldap.LdapCtxFactory"
 -j "java.naming.provider.url=
            ldap://mydomain.com:389/o=imq"
 -j "java.naming.security.principal=
            uid=fooUser, ou=People, o=imq"
 -j "java.naming.security.credentials=fooPasswd"
 -j "java.naming.security.authentication=simple"
```

Chapter 8

Managing Security

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. You can use a different user repository for each broker instance. The first part of this chapter explains how you create, populate, and manage that repository. For an introduction to Sun™ ONE Message Queue (MQ) security, see "Security Manager" on page 66.

Authorizing Users: the Access Control Properties File You are responsible for editing an access control properties file that maps each user's access to broker operations to the user's name or group membership. You can use a different access control properties file for each broker instance. The second part of this chapter explains how you can customize this file.

Encryption: Working With an SSL-based Service (Enterprise Edition) Using a connection service based on the Secure Socket Layer (SSL) standard allows you to encrypt messages sent between clients and broker. For an introduction to how MQ handles encryption, see "Encryption (Enterprise Edition)" on page 68. The last part of this chapter explains how to set up an SSL-based connection service and provides additional information about using SSL.

For situations in which a password is needed for a broker to secure access to an SSL keystore, an LDAP user repository, or a JDBC-compliant persistent store, there are three means of providing such passwords:

- by having the system prompt you when the broker is started
- by passing in passwords as command line options when starting the broker (see "Starting a Broker" on page 134 and Table 5-2 on page 136)
- by storing passwords in a passfile that the system accesses when starting the broker (See "Using a Passfile" on page 224)

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 broker-specific user repository that each broker is configured to consult. This repository can be of two types:

- a flat-file repository that is shipped with MQ

  This type of user repository is very easy to use; however it is vulnerable to security attacks, and should therefore be used *only* for evaluation and development purposes. You can populate and manage the repository using the User Manager utility (`imqusermgr`). To enable authentication, you populate the user repository with each user's name, password, and the name of the user's group.

  For more information on setting up and managing the user repository, see "Using a Flat-File User Repository."

- an LDAP server

  This could be an existing or new LDAP directory server that uses the LDAP v2 or v3 protocol. It is not as easy to use as the flat-file repository, however it is secure and scalable, and therefore better for production environments.

  If you are using an LDAP user repository, you will need to use the tools provided by the LDAP vendor to populate and manage the user repository. For more information, see "Using an LDAP Server for a User Repository" on page 209.

Using a Flat-File User Repository

MQ provides a flat-file user repository and a command line tool, MQ User Manager (`imqusermgr`) that you can use to populate and manage the flat-file user repository. The following sections describe the flat-file user repository and how you use the MQ User Manager utility (`imqusermgr`) to populate and manage that repository.

Creating a User Repository

The flat-file user repository is instance specific. A default user repository (named `passwd`) is created for each broker instance that you start. This user repository is placed in a directory identified by the name of the broker instance (*instanceName*) with which the repository is associated (see Appendix A, "Location of MQ Data"):

…/`instances/`*instanceName*`/etc/passwd`

The repository is created with two entries (rows), as illustrated in Table 8-1, below.

Table 8-1 Initial Entries in User Repository

User Name	Password	Group	State
`admin`	`admin`	`admin`	`active`
`guest`	`guest`	`anonymous`	`active`

These initial entries allow the MQ broker to be used immediately after installation without any intervention by the administrator. In other words, no initial user/password setup is required for the MQ broker to be used.

The initial `guest` user entry allows clients to connect to a broker instance using the default `guest` user name and password (for testing purposes, for example).

The initial `admin` user entry allows you to use `imqcmd` commands to administer a broker instance using the default `admin` user name and password. It is recommended that you update this initial entry to change the password (see "Changing the Default Administrator Password" on page 208).

The following sections explain how you populate and manage a flat-file user repository.

User Manager Utility (imqusermgr)

The User Manager utility (`imqusermgr`) lets you edit or populate a flat-file user repository.

This section describes the basic `imqusermgr` command syntax, provides a listing of subcommands, and summarizes `imqusermgr` command options. Subsequent sections explain how you use the `imqusermgr` subcommands to accomplish specific tasks.

Before using `imqusermgr`, keep the following things in mind:

- *If* a broker-specific user repository does not yet exist, you have to start up the corresponding broker instance to create it.
- The `imqusermgr` command has to be run on the host where the broker is installed
- You have to have the appropriate permissions to write to the repository,: namely, on Solaris and Linux, you have to be the root user or the user who first created the broker instance

Syntax of the imqusermgr Command

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

```
imqusermgr subcommand [options]
imqusermgr -h
imqusermgr -v
```

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

```
imqusermgr list -v
```

imqusermgr Subcommands

Table 8-2 lists the `imqusermgr` subcommands.

Table 8-2 `imqusermgr` Subcommands

Subcommand	Description
`add [-i` *instanceName*`] -u` *userName* `-p` *passwd* `[-g` *group*`] [-s]`	Adds a user and associated password to the specified (or default) broker instance repository, and optionally specifies the user's group.
`delete [-i` *instanceName*`] -u` *userName* `[-s] [-f]`	Deletes the specified user from the specified (or default) broker instance repository.
`list [-i` *instanceName*`] [-u` *userName*`]`	Displays information about the specified user or all users in the specified (or default) broker instance repository.
`update [-i` *instanceName*`] -u` *userName* `-p` *passwd* `[-a` *state*`] [-s] [-f]` `update [-i` *instanceName*`] -u` *userName* `-a` *state* `[-p` *passwd*`] [-s] [-f]`	Updates the password and/or state of the specified user in the specified (or default) broker instance repository.

NOTE Examples in the following sections assume the default broker instance.

Summary of imqusermgr Command Options

Table 8-3 lists the options to the `imqusermgr` command.

Table 8-3 `imqusermgr` Options

Option	Description
`-a` *active_state*	Specifies (`true`/`false`) whether the user's state should be active. A value of `true` means that the state is active. This is the default.
`-f`	Performs action without user confirmation
`-h`	Displays usage help. Nothing else on the command line is executed.
`-i` *instanceName*	Specifies the broker instance user repository to which the command applies. If not specified, the default *instanceName*, `imqbroker`, is assumed.
`-p` *passwd*	Specifies the user's password.
`-g` *group*	Specifies the user group. Valid values are `admin`, `user`, `anonymous`.
`-s`	Sets silent mode.
`-u` *userName*	Specifies the user name.
`-v`	Displays version information. Nothing else on the command line is executed.

Groups

When adding a user entry to the user repository for a broker instance, you have 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.

- **`admin` group**. For broker administrators. Users who are assigned this group can, by default, configure, administer, and manage the broker. You can assign more than one user to the `admin` group.

- **`user` group**. For normal (non-administrtion) MQ client users. Most client users will access the broker by being authenticated in the `user` group. By default, application client users in this group can produce messages to and consume messages from all topics and queues, or can browse messages in any queue by default.
- **`anonymous` group**. For MQ clients that do not wish to use a user name that is known to the broker (possibly because the client application does not know of a real user name to use). This is analogous to the anonymous account present in most FTP servers. You 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 users from this group at deployment time.

In order to change a user's group, you 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 212.

States

When you add a user to a repository, the user's state is active by default. To make the user inactive, you must use the update command. For example, the following command makes the user `JoeD` inactive:

```
imqusermgr update -u JoeD -a false
```

Entries for users that have been rendered inactive are retained in the repository; however, inactive users cannot open new connections. If a user is inactive and you add another user who has the same name, the operation will fail. You must delete the inactive user entry or change the new user's name or use a different name for the new user. This prevents you from adding duplicate user names.

Format of User Names and Passwords

User names and passwords must follow these guidelines:

- The user name 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 a User Repository

Use the `add` subcommand to add a user to a repository. For example, the following command adds the user, `Katharine` with the password `sesame` to the default broker instance user repository.

```
imqusermgr add -u Katharine -p sesame -g user
```

Use the `delete` subcommand to delete a user from a repository. For example, the following command deletes the user, `Bob`:

```
imqusermgr delete -u Bob
```

Use the `update` subcommand to change a user's password or state. For example, the following command changes Katharine's password to `alladin`:

```
imqusermgr update -u Katharine -p alladin
```

To list information about one user or all users, use the `list` command. The following command shows information about the user named `isa`:

```
imqusermgr list -u isa
```

```
% imqusermgr list -u isa

User repository for broker instance: imqbroker
---------------------------------
User Name    Group    Active State
---------------------------------
isa          admin    true
```

The following command lists information about all users:

```
imqusermgr list
```

```
% imqusermgr list

User repository for broker instance: imqbroker
-------------------------------------
User Name    Group        Active State
-------------------------------------
admin        admin        true
guest        anonymous    true
isa          admin        true
testuser1    user         true
testuser2    user         true
testuser3    user         true
testuser4    user         false
testuser5    user         false
```

Changing the Default Administrator Password

For the sake of security, you should change the default password of `admin` to one that is only known to you. You need to use the `imqusermgr` tool to do this.

The following command changes the default password for the `mybroker` broker instance to `grandpoobah`.

```
imqusermgr update -i mybroker -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 instance is running. For example, the following command should work,

```
imqcmd list svc -i mybroker -u admin -p grandpoobah
```

While using the old password should fail.

After changing the password, you should supply the new password any time you use any of the MQ administration tools, including the Administration Console.

Using an LDAP Server for a User Repository

If you want to use an LDAP server for a user repository, you must set certain broker properties in the instance configuration file. These properties enable the broker instance to query the LDAP server for information about users and groups whenever a user attempts to connect to the broker instance or perform certain messaging operations. The instance configuration file (`config.properties`) is located in a directory identified by the name of the broker instance (*instanceName*) with which the configuration file is associated (see Appendix A, "Location of MQ Data"):

...`/instances/`*instanceName*`/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:

   ```
   imq.authentication.basic.user_repository=ldap
   ```

2. Set the `imq.authentication.type` property to determine whether a password should be passed from client to broker in base64 encoding (`basic`) or in MD5 digest (`digest`). When using an LDAP directory server for a user repository, you must set the authentication type to `basic`. For example,

   ```
   imq.authentication.type=basic
   ```

3. You must also set the broker properties that control LDAP access. These properties, stored in a broker's instance configuration file, are described in Table 8-5. MQ uses JNDI APIs to communicate with the LDAP directory server. Consult JNDI documentation for more information on syntax and on terms referenced in these properties. MQ uses a Sun JNDI LDAP provider and uses simple authentication.

MQ supports LDAP authentication failover: you can specify a list of LDAP directory servers for which authentication will be attempted (see the `imq.user.repos.ldap.server` property in Table 8-5).

Table 8-5 LDAP-related Properties

Property	Description
`imq.user_repository.ldap.server`	The *host:port* for the LDAP server, where *host* specifies the fully qualified DNS name of the host running the directory server and *port* specifies the port number that the directory server is using for communications. To specify a list of failover servers, use the following syntax: *host1:port1* `ldap://`*host2:port2* `ldap://`*host3:port3*... where entries in the list are separated by spaces. Note that each failover server address after the first one begins with `ldap`.
`imq.user_repository.ldap.principal`	The distinguished name that the broker will use to bind to the directory server for a search. If the directory server allows anonymous searches, this property does not need to be assigned a value.
`imq.user_repository.ldap.password`	The password associated with the distinguished name used by the broker. This property can only be specified in a passfile (see "Using a Passfile" on page 224). There are a number of ways to provide a password. The most secure is to let the broker prompt you for a password. Less secure is to use a passfile and read-protect the passfile. Least secure is to specify the password using the following command line option: `imqbrokerd -ldappassword`. If the directory server allows anonymous searches, no password is needed.
`imq.user_repository.ldap.base`	The directory base for user entries.
`imq.user_repository.ldap.uidattr`	The provider-specific attribute identifier whose value uniquely identifies a user. For example: `uid`, `cn`, etc.
`imq.user_repository.ldap.usrfilter`	A JNDI search filter (a search query expressed as a logical expression). By specifying a search filter for users, the broker can narrow the scope of a search and thus make it more efficient. For more information, see the JNDI tutorial at the following location: `http://java.sun.com/products/jndi/tutorial`. This property does not have to be set.

Table 8-5 LDAP-related Properties

Property	Description
`imq.user_repository.ldap.grpsearch`	A boolean specifying whether you want to enable group searches. Consult the documentation provided by your LDAP provider to determine whether you can associate users into groups. Note that nested groups are not supported in MQ. Default: `false`
`imq.user_repository.ldap.grpbase`	The directory base for group entries.
`imq.user_repository.ldap.gidattr`	The provider-specific attribute identifier whose value is a group name.
`imq.user_repository.ldap.memattr`	The attribute identifier in a group entry whose values are the distinguished names of the group's members.
`imq.user_repository.ldap.grpfiltler`	A JNDI search filter (a search query expressed as a logical expression). By specifying a search filter for groups, the broker can narrow the scope of a search and thus make it more efficient. For more information, see the JNDI tutorial at the following location. `http://java.sun.com/products/jndi/tutorial` This property does not have to be set.
`imq.user_repository.ldap.timeout`	An integer specifying (in seconds) the time limit for a search. By default this is set to 180 seconds.
`imq.user_repository.ldap.ssl.enabled`	A boolean specifying whether the broker should use the SSL protocol when talking to an LDAP server. This is set to `false` by default.

See the broker's default.properties file for a sample (default) LDAP user-repository-related properties setup.

4. If necessary, you need to edit the users/groups and rules in the access control properties file. For more information about the use of access control property files, see "Authorizing Users: the Access Control Properties File" on page 212.

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 the port used by the LDAP server for SSL communications. For example:

```
imq.user_repository.ldap.server=myhost:7878
```

- Set the broker property `imq.user_repository.ldap.ssl.enabled` to `true`.

Authorizing Users: the Access Control Properties File

After connecting to a broker instance, a user might 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 a broker-specific *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 ACL file to restrict these operations to certain users and groups.

The ACL file is used independently of whether user information is placed in a flat-file user repository (see "Using a Flat-File User Repository" on page 202) or in an LDAP user repository (see "Using an LDAP Server for a User Repository" on page 209).

Creating an Access Control Properties File

The ACL file is instance specific. A default file (named `accesscontrol.properties`) is created for each broker instance that you start. This ACL properties file is placed in a directory identified by the name of the broker instance (*instanceName*) with which the ACL file is associated (see Appendix A, "Location of MQ Data"):

…`/instances/`*instanceName*`/etc/accesscontrol.properties`

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 205.
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/1.4/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 `q1`:

  ```
  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-based Service (Enterprise Edition)

The MQ Enterprise Edition supports connection services based on the Secure Socket Layer (SSL) standard: over TCP/IP (`ssljms` and `ssladmin`) and over HTTP (`httpsjms`). These SSL-based connection services allow for the encryption of messages sent between clients and broker. The current MQ release supports SSL encryption based on self-signed server certificates.

To use an SSL-based connection service, you need to generate a private key/public key pair using the Key Tool utility (`imqkeytool`). This utility embeds the public key in a self-signed certificate that is passed to any client requesting a connection to the broker, and the client uses the certificate to set up an encrypted connection.

While MQ's SSL-based connection services are similar in concept, there are some differences in how you set them up. Secure connections over TCP/IP and over HTTP are therefore discussed separately in the following sections.

Setting Up an SSL-based Service Over TCP/IP

There are three SSL-based connection services that provide a direct, secure connection over TCP/IP.

ssljms This connection service is used to deliver messages over a secure, encrypted connection between a client and broker.

ssladmin This connection service is used to create a secure, encrypted connection between the MQ Command utility (`imqcmd`)—the command line administration tool—and a broker. A secure connection is not supported for the Administration Console (`imqadmin`).

cluster This connection service is used to deliver messages over a secure, encrypted connection between brokers in a cluster (see "Secure Inter-Broker Connections" on page 143).

➤ **To Set Up an SSL-based Connection Service**

1. Generate a self-signed certificate.
2. Enable the ssljms connection service, ssladmin connection service, or cluster connection service in the broker.
3. Start the broker.
4. Configure and run the client (applies only to `ssljms` connection service).

The procedures for setting up ssljms and ssladmin connection services are identical, except for Step 4, configuring and running the client.

Each of the steps is discussed in some detail in the sections that follow.

Step 1. Generating a Self-Signed Certificate

MQ's SSL Support is oriented toward securing on-the-wire data with the assumption that the client is communicating with a known and trusted server. Therefore, SSL is implemented using only self-signed certificates.

Run the `imqkeytool` command to generate a self-signed certificate for the broker. The same certificate can be used for the ssljms connection service, ssladmin connection service, or cluster connection service. Enter the following at the command prompt:

```
imqkeytool -broker
```

The utility will prompt you for the information it needs. (On UNIX® systems you may need to run `imqkeytool` as the superuser (`root`) in order to have permission to create the keystore.)

First, `imqkeytool` prompts you for a keystore password, then it prompts you for some organizational information, and then it prompts you for confirmation. After it receives the confirmation, it pauses while it generates a key pair. It then asks you for a password to lock the particular key pair (key password); you should enter Return in response to this prompt: this makes the key password the same as the keystore password.

NOTE Remember the password you provide—you will need to provide this password later to the broker (when you start it) so it can open the keystore. You can also store the keystore password in a passfile (see "Using a Passfile" on page 224).

Running `imqkeytool` runs the JDK `keytool` utility to generate a self-signed certificate and places it in MQ's keystore, located in a directory that depends upon the operating system, as shown in Appendix A, "Location of MQ Data."

The keystore is in the same format as that supported by the JDK1.2 `keytool` utility.

The configurable properties for the MQ keystore are shown in Table 8-8. (For instructions on configuring these properties, see Chapter 5, "Starting and Configuring a Broker.")

Table 8-8 Keystore Properties

Property Name	Description
`imq.keystore.file.dirpath`	For SSL-based services: specifies the path to the directory containing the keystore file. Default: see Appendix A, "Location of MQ Data."
`imq.keystore.file.name`	For SSL-based services: specifies the name of the keystore file. Default: `keystore`
`imq.keystore.password`	For SSL-based services: specifies the keystore password. This property can only be specified in a passfile (see "Using a Passfile" on page 224). There are a number of ways to provide a password. The most secure is to let the broker prompt you for a password. Less secure is to use a passfile and read-protect the passfile. Least secure is to specify the password using the following command line option: `imqbrokerd -ldappassword`.

You may need to regenerate a key pair in order to solve certain problems; for example:

- You forgot the keystore password.
- The SSL-based service fails to initialize when you start a broker and you get the exception:
  `java.security.UnrecoverableKeyException: Cannot recover key`.

  This exception may result from the fact that you had provided a key password that was different from the keystore password when you generated the self-signed certificate in "Step 1. Generating a Self-Signed Certificate" on page 219.

➤ **To Regenerate a Key Pair**

1. Remove the broker's keystore, located as shown in Appendix A, "Location of MQ Data."
2. Rerun `imqkeytool` to generate a key pair as described in "Step 1. Generating a Self-Signed Certificate" on page 219.

Step 2. Enabling the SSL-based Service in the Broker

To enable the SSL-based service in the broker, you need to add `ssljms` (or `ssladmin`) to the `imq.service.activelist` property.

NOTE The SSL-based `cluster` connection service is enabled using the `imq.cluster.transport` property rather than the `imq.service.activelist` property. See "Secure Inter-Broker Connections" on page 143.

➤ **To Enable an SSL-based Service in the Broker**

1. Open the broker's instance configuration file.

   The instance configuration file is located in a directory identified by the name of the broker instance (*instanceName*) with which the configuration file is associated (see Appendix A, "Location of MQ Data"):

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

2. Add an entry (if one does not already exist) for the `imq.service.activelist` property and include SSL-based services in the list.

   By default, the property includes the jms and admin connection services. You need to add the ssljms or ssladmin connection services or both (depending on the services you want to activate):

   ```
   imq.service.activelist=jms,admin,ssljms,ssladmin
   ```

Step 3. Starting the Broker

Start the broker, providing the keystore password. You can provide the password in any one of the following ways:

- Allow the broker to prompt you for the password when it starts up

  ```
  imqbrokerd
  Please enter Keystore password: mypassword
  ```

- Use the `-password` option to the `imqbrokerd` command:

  ```
  imqbrokerd -password mypassword
  ```

- Put the password in a passfile file (see "Using a Passfile" on page 224) which is accessed at broker startup. You have to first set the following broker configuration property (see "Editing the Instance Configuration File" on page 129):

  `imq.passfile.enabled=true`

  Once this property is set, you can access the passfile in either of two ways:

  - pass the location of the passfile to the `imqbrokerd` command:

    `imqbrokerd -passfile` *`/tmp/mypassfile`*

  - start the broker without the -passfile option, but specify the location of the passfile using the following two broker configuration properties:

    `imq.passfile.dirpath=`*`/tmp`*

    `imq.passfile.name=`*`mypassfile`*

For a listing of passfile-related broker properties, see Table 2-6 on page 69.

When you start a broker or client with SSL, you might notice that it consumes a lot of cpu cycles for a few seconds. This is because MQ uses JSSE (Java Secure Socket Extension) to implement SSL. JSSE uses `java.security.SecureRandom()` to generate random numbers. This method takes a significant amount of time to create the initial random number seed, and that is why you are seeing increased cpu usage. After the seed is created, the cpu level will drop to normal.

Step 4. Configuring and Running SSL-based Clients

Finally, you need to configure clients to use the secure connection services. There are two types of clients, depending on the connection service you are using: application clients that use `ssljms`, and the MQ administration clients (such as `imqcmd`) that use `ssladmin`. These are treated separately in the following sections.

Application Clients

You have to make sure the client has the necessary Secure Socket Extension (JSSE) jar files in its classpath, and you need to tell it to use the `ssljms` connection service.

1. If your client is not using J2SDK1.4 (which has JSSE and JNDI support built in), make sure the client has the following jar files in its class path:

   `jsse.jar, jnet.jar, jcert.jar, jndi.jar`

2. Make sure the client has the following MQ jar files in its class path:

   imq.jar, jms.jar

3. Start the client and connect to the broker's ssljms service. One way to do this is by entering a command like the following:

   `java -DimqConnectionType=TLS` *clientAppName*

   Setting `imqConnectionType` tells the connection to use SSL.

   For more information on using `ssljms` connection services in client applications, see the chapter on using administered objects in the MQ *Java Client Developer's Guide*.

Administration Clients (imqcmd)

You can establish a secure administration connection by including the `-secure` option when using `imqcmd` (see Table 6-2 on page 154) for example:

imqcmd list svc -b *hostName:port* -u *adminName* -p *adminPassword* -secure

where *adminName* and *adminPassword* are valid entries in the MQ user repository. (If you are using a flat-file repository, see "Changing the Default Administrator Password" on page 208).

Listing the connection services is a way to show that the `ssladmin` service is running, and that you can successfully make a secure admin connection, as shown in the following output:

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

Host                 Primary Port
localhost            7676

Service Name     Port Number       Service State
admin            33984 (dynamic)   RUNNING
httpjms          -                 UNKNOWN
httpsjms         -                 UNKNOWN
jms              33983 (dynamic)   RUNNING
ssladmin         35988 (dynamic)   RUNNING
ssljms           dynamic           UNKNOWN

Successfully listed services.
```

Setting Up an SSL Service Over HTTP

In this SSL-based connection service (httpsjms), the client and broker establish a secure connection by way of a HTTPS tunnel servlet. The architecture and implementation of HTTPS support is described in Appendix C, "HTTP/HTTPS Support (Enterprise Edition)" on page 305.

Using a Passfile

In cases where you want the broker to start up without prompting you for needed passwords, or without requiring you to supply these passwords as options to the `imqbrokerd` command, you can place the needed passwords in a *passfile*. This passfile can then be specified using the `-passfile` option when starting up a broker:

```
imqbrokerd -passfile myPassfile
```

A passfile is a simple text file containing passwords. The file is not encrypted, and therefore less secure than supplying passwords manually. Nevertheless you can set permissions on the file that limit who has access to view it. The permissions set on the passfile need to provide read access to the user who starts the broker.

A passfile can contain the passwords shown in Table 8-9:

Table 8-9 Passwords in a Passfile

Password	Description
`imq.keystore.password`	Specifies the keystore password for SSL-based services (see Table 8-8 on page 220).
`imq.user_repository.ldap.password`	Specifies the password associated with the distinguished name assigned to a broker for binding to a configured LDAP user repository (see Table 8-5 on page 210).
`imq.persist.jdbc.password`	Specifies the password used to open a database connection, if required (see Table B-1 on page 298).

A sample passfile can be found at a location that depends on operating system, as shown in Appendix A, "Location of MQ Data."

Chapter 9

Analyzing and Tuning an MQ Message Service

This chapter covers a number of topics about how to analyze and tune an MQ message service to optimize the performance of your messaging applications. It includes the following topics:

- About Performance
- Factors That Impact Performance
- Monitoring an MQ Message Server
- Troubleshooting Performance Problems
- Adjusting Your Configuration To Improve Performance

About Performance

The Performance Tuning Process

The performance you get out of a messaging application depends on the interaction between the application and the MQ message service. Hence, maximizing performance requires the combined efforts of both the application developer and the MQ administrator.

the process of optimizing performance begins with application design and continues through to tuning the message service after the application has been deployed. The performance tuning process includes the following stages:

- Defining performance requirements for the application

- Designing the application taking into account factors that affect performance (especially trade-offs between reliability and performance)
- Establishing baseline performance measures
- Tuning or reconfiguring the MQ message service to optimize performance.

The process outlined above is often iterative. During deployment of the application, an MQ administrator evaluates the suitability of the message server for the application's general performance requirements. If the benchmark testing meets these requirements, the administrator can tune the system as described in this chapter. However, if benchmark testing does not meet performance requirements, then a redesign of the application might be necessary or the deployment architecture might need to be modified.

Aspects of Performance

In general, performance is a measure of the speed and efficiency with which a message service delivers messages from producer to consumer. However, there are several different aspects of performance that might be important to you, depending on your needs.

Connection Load The number of message producers, or message consumers, or the number of concurrent connections a system can support.

Message throughput The number of messages or message bytes that can be pumped through a messaging system per second.

Latency The time it takes a particular message to be delivered from message producer to message consumer.

Stability The overall availability of the message service or how gracefully it degrades in cases of heavy load or failure.

Efficiency The efficiency of message delivery; a measure of message throughput in relation to the computing resources employed.

These different aspects of performance are generally inter-related. If message throughput is high, that means messages are less likely to be backlogged in the message server, and as a result, latency should be low (a single message can be delivered very quickly). However, latency can depend on many factors: the speed of communication links, message server processing speed, and client processing speed, to name a few.

In any case, there are several different aspects of performance. Which of them are most important to you generally depends on the requirements of a particular application.

Benchmarks

Benchmarking is the process of creating a test suite for your messaging application and of measuring message throughput or other aspects of performance for this test suite.

For example, you could create a test suite by which some number of producing clients, using some number of connections, sessions, and message producers, send persistent or non-persistent messages of a standard size to some number of queues or topics (all depending on your messaging application design) at some specified rate. Similarly, the test suite includes some number of consuming clients, using some number of connections, sessions, and message consumers (of a particular type) that consume the messages in the test suite's destinations using a particular acknowledgement mode.

Using your standard test suite you can measure the time it takes between production and consumption of messages or the average message throughput rate, and you can monitor the system to observe connection thread usage, message storage data, message flow data, and other relevant metrics. You can then ramp up the rate of message production, or the number of message producers, or other variables, until performance is negatively impacted. The maximum throughput you can achieve is a benchmark for your message service configuration.

Using this benchmark, you can modify some of the characteristics of your test suite. By carefully controlling all the factors that might have an impact on performance (see "Application Design Factors that Impact Performance" on page 230), you can note how changing some of these factors affects the benchmark. For example, you can increase the number of connections or the size of messages five-fold or ten-fold, and note the impact on performance.

Conversely, you can keep application-based factors constant and change your broker configuration in some controlled way (for example, change connection properties, thread pool properties, JVM memory limits, limit behaviors, built-in versus plugged-in persistence, and so forth) and note how these changes affect performance.

This benchmarking of your application provides information that can be valuable when you want to increase the performance of a deployed application by tuning your message service. A benchmark allows the effect of a change or a set of changes to be more accurately predicted.

As a general rule, benchmarks should be run in a controlled test environment and for a long enough period of time for your message service to stabilize. (Performance is negatively impacted at startup by the Just-In-Time compilation that turns Java code into machine code.)

Baseline Use Patterns

Once a messaging application is deployed and running, it is important to establish baseline use patterns. You want to know when peak demand occurs and you want to be able to quantify that demand. For example, demand normally fluctuates by number of end-users, activity levels, time of day, or all of these.

To establish base-line use patterns you need to monitor your message server over an extended period of time, looking at data such as number of connections, number of messages stored in the broker (or in particular destinations), message flows into and out of a broker (or particular destinations), numbers of active consumers, and so forth. You can also use average and peak values provided in metrics data.

It is important to check these baseline metrics against design expectations. By doing so, you are checking that client code is behaving properly: for example, that connections are not being left open or that consumed messages are not being left unacknowledged. These coding errors consume message server resources and could significantly affect performance.

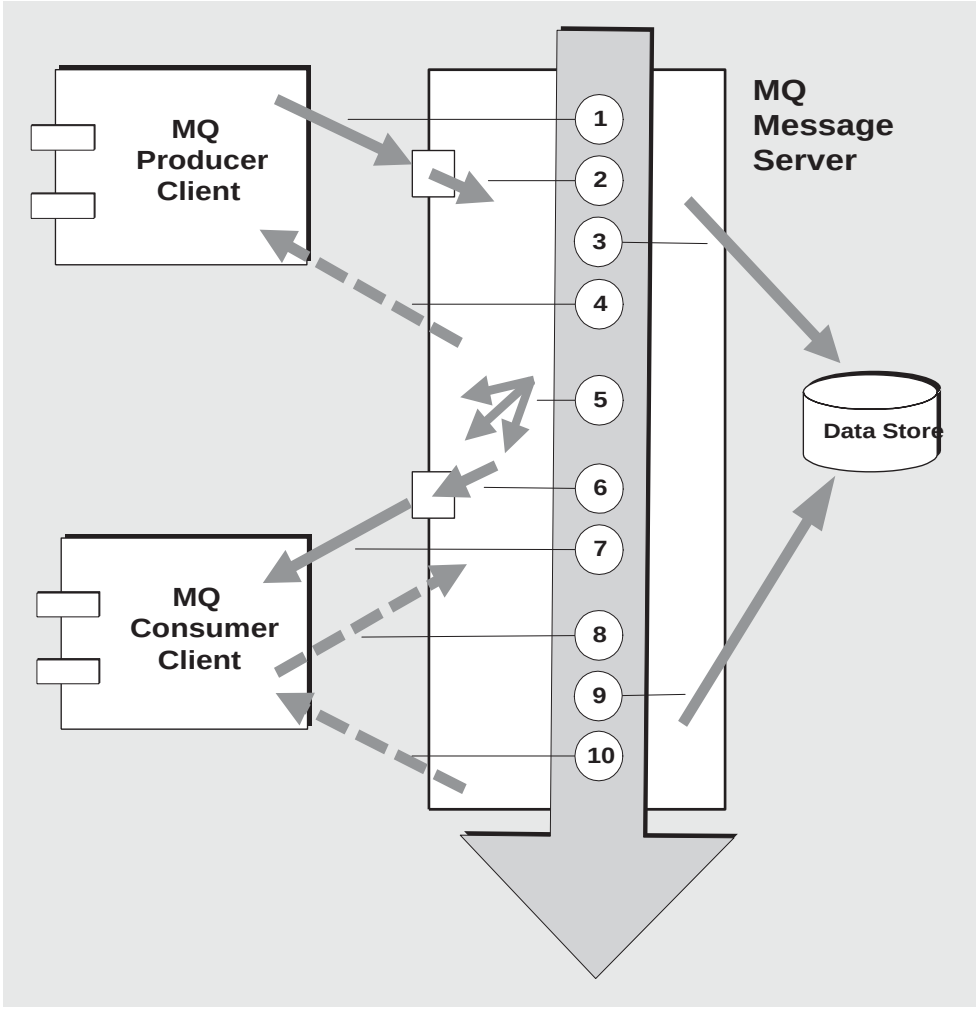
The base-line use patterns help you determine how to tune your system for optimal performance. For example, if one destination is used significantly more than others, you might want to set higher message memory limits on that destination than on others, or to adjust limit behaviors accordingly. If the number of connections needed is significantly greater than allowed by the maximum thread pool size, you might want to increase the threadpool size or adopt a shared thread model. If peak message flows are substantially greater than average flows, that might influence the limit behaviors you employ when memory runs low.

In general, the more you know about use patterns, the better you are able to tune your system to those patterns and to plan for future needs.

Factors That Impact Performance

Message latency and message throughput, two of the main performance indicators, generally depend on the time it takes a typical message to complete various steps in the message delivery process. These steps are shown below for the case of a persistent, reliably delivered message. The steps are described following the illustration.

Figure 9-1 Message Delivery Through an MQ System



1. The message is delivered from producing client to message server
2. The message server reads in the message
3. The message is placed in persistent storage (for reliability)
4. The message server confirms receipt of the message (for reliability)
5. The message server determines the routing for the message
6. The message server writes out the message
7. The message is delivered from message server to consuming client
8. The consuming client acknowledges receipt of the message (for reliability)
9. The message server processes client acknowledgement (for reliability)
10. The message server confirms that client acknowledgement has been processed

Since these steps are sequential, any one of them can be a potential bottleneck in the delivery of messages from producing clients to consuming clients. Most of these steps depend upon physical characteristics of the messaging system: network bandwidth, computer processing speeds, message server architecture, and so forth. Some, however, also depend on characteristics of the messaging application and the level of reliability it requires.

The following subsections discuss the impact of both application design factors and messaging system factors on performance. While application design and messaging system factors closely interact in the delivery of messages, each category is considered separately.

Application Design Factors that Impact Performance

Application design decisions can have a significant effect on overall messaging performance.

The most important factors affecting performance are those that impact the reliability of message delivery. Among these are the following factors:

- Delivery Mode (Persistent/Non-persistent Messages)
- Use of Transactions
- Acknowledgement Mode
- Durable vs. Non-Durable Subscriptions

Other application design factors impacting performance are the following:

- Use of Selectors (Message Filtering)
- Message Size
- Message Body Type

The sections that follow describe the impact of each of these factors on messaging performance. As a general rule, there is a trade-off between performance and reliability: factors that increase reliability tend to decrease performance.

The following table shows how the various application design factors generally affect messaging performance. The table shows two scenarios—a high reliability, low performance scenario and a high performance, low reliability scenario—and the choice of application design factors that characterizes each. Between these extremes, there are many choices and trade-offs that affect both reliability and performance.

Table 9-1 Comparison of High Reliability and High Performance Scenarios

Application Design Factor	High Reliability Low Performance Scenario	High Performance Low Reliability Scenario
Delivery mode	Persistent messages	Non-persistent messages
Use of transactions	Transacted sessions	No transactions
Acknowledgement mode	`AUTO_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE`	`DUPS_OK_ACKNOWLEDGE`
Durable/non-durable subscriptions	Durable subscriptions	Non-durable subscriptions
Use of selectors	Message filtering	No message filtering
Message size	Small messages	Large messages
Message body type	Complex body types	Simple body types

NOTE In the graphs that follow, performance data were generated on a two-CPU, 1002 Mhz, Solaris 8 system, using file-based persistence. The performance test first warmed up the Message Queue broker, allowing the Just-In-Time compiler to optimize the system and the persistent database to be primed.

Once the broker was warmed up, a single producer and single consumer were created and messages were produced for 30 seconds. The time required for the consumer to receive all produced messages was recorded, and a throughput rate (messages per second) was calculated. This scenario was repeated for different combinations of the application design factors shown in Table 9-1.

Delivery Mode (Persistent/Non-persistent Messages)

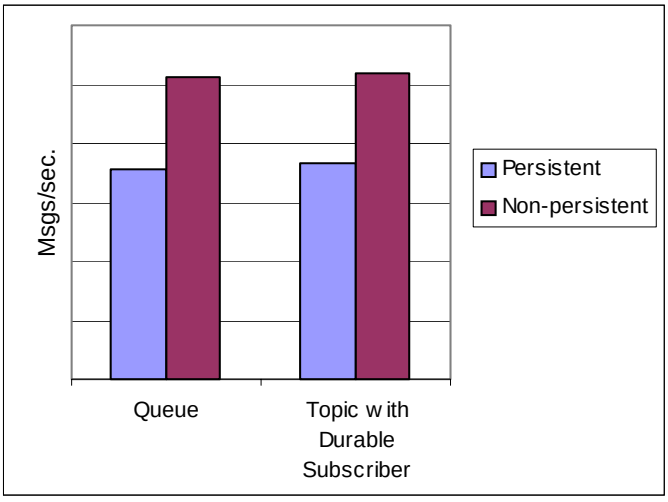
As described in "Reliable Messaging" on page 45, persistent messages guarantee message delivery in case of message server failure. The broker stores the message in a persistent store until all intended consumers acknowledge they have consumed the message.

Broker processing of persistent messages is slower than for non-persistent messages for the following reasons:

- A broker must reliably store a persistent message so that it will not be lost should the broker fail.
- The broker must confirm receipt of each persistent message it receives. Delivery to the broker is guaranteed once the method producing the message returns without an exception.
- Depending on the client acknowledgment mode, the broker might need to confirm a consuming client's acknowledgement of a persistent message.

The differences in performance between the persistent and non-persistent modes can be significant. Figure 9-2 compares throughput for persistent and non-persistent messages in two reliable delivery cases: 10k-sized messages delivered both to a queue and to a topic with durable subscriptions. Both cases use the `AUTO_ACKNOWLEDGE` acknowledgement mode.

Figure 9-2 Performance Impact of Delivery Modes



Use of Transactions

A transaction is a guarantee that all messages produced in a transacted session and all messages consumed in a transacted session will be either processed or not processed (rolled back) as a unit.

MQ supports both local and distributed transactions (see "Local Transactions" and "Distributed Transactions" on page 46, respectively, for more information).

A message produced or acknowledged in a transacted session is slower than in a non-transacted session for the following reasons:

- Additional information must be stored with each produced message.
- In some situations, messages in a transaction are stored when normally they would not be (for example, a persistent message delivered to a topic destination with no subscriptions would normally be deleted, however, at the time the transaction is begun, information about subscriptions is not available).
- Information on the consumption and acknowledgement of messages within a transaction must be stored and processed when the transaction is committed.

Acknowledgement Mode

One mechanism for ensuring the reliability of JMS message delivery is for a client to acknowledge consumption of messages delivered to it by the MQ message server (see "Reliable Delivery: Acknowledgements and Transactions" on page 59)

If a session is closed without the client acknowledging the message or if the MQ message server fails before the acknowledgment is processed, the broker redelivers that message, setting a `JMSRedelivered` flag.

For a non-transacted session, the client can choose one of three acknowledgement modes, each of which has its own performance characteristics:

- `AUTO_ACKNOWLEDGE`. The system automatically acknowledges a message once the consumer has processed it. This mode guarantees at most one redelivered message after a provider failure.
- `CLIENT_ACKNOWLEDGE`. The application controls the point at which messages are acknowledged. All messages processed in that session since the previous acknowledgement are acknowledged. If the message server fails while processing a set of acknowledgments, one or more messages in that group might be redelivered.
- `DUPS_OK_ACKNOWLEDGE`. This mode instructs the system to acknowledge messages in a lazy manner. Multiple messages can be redelivered after a provider failure.

(Using `CLIENT_ACKNOWLEDGE` mode is similar to using transactions, except there is no guarantee that all acknowledgments will be processed together if a provider fails during processing.)

Performance is impacted by acknowledgement mode for the following reasons:

- Extra control messages between broker and client are required in `AUTO_ACKNOWLEDGE` and `CLIENT_ACKNOWLEDGE` modes. The additional control messages add additional processing overhead and can interfere with JMS payload messages, causing processing delays.
- In `AUTO_ACKNOWLEDGE` and `CLIENT_ACKNOWLEDGE` modes, the client must wait until the broker confirms that it has processed the client's acknowledgment before the client can consume additional messages. (This broker confirmation guarantees that the broker will not inadvertently redeliver these messages.)
- The MQ persistent store must be updated with the acknowledgement information for all persistent messages received by consumers, thereby decreasing performance.

Durable vs. Non-Durable Subscriptions

Subscribers to a topic destination fall into two categories, those with durable and non-durable subscriptions, as described in "Publish/Subscribe (Topic destinations)" on page 43:

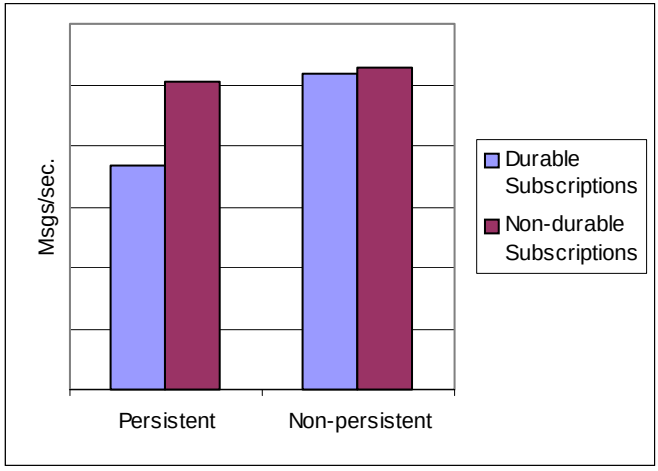
Durable subscriptions provide increased reliability at the cost of slower throughput for the following reasons:

- The MQ message server must persistently store the list of messages assigned to each durable subscription so that should a message server fail, the list is available after recovery.
- Persistent messages for durable subscriptions are stored persistently, so that should a message server fail, the messages can still be delivered after recovery, when the corresponding consumer becomes active. By contrast, persistent messages for non-durable subscriptions are not stored persistently (should a message server fail, the corresponding consumer connection is lost and the message would never be delivered).

Figure 9-3 compares throughput for topic destinations with durable and non-durable subscriptions in two cases: persistent and non-persistent 10k-sized messages. Both cases use `AUTO_ACKNOWLEDGE` acknowledgement mode.

You can see from Figure 9-3 that the performance impact of using durable subscriptions is manifest only in the case of persistent messages; and the impact in that case is because persistent messages are only stored persistently for durable subscriptions, as explained above.

Figure 9-3 Performance Impact of Subscription Types



Use of Selectors (Message Filtering)

Application developers often want to target sets of messages to particular consumers. They can do so either by targeting each set of messages to a unique destination or by using a single destination and registering one or more selectors for each consumer.

A selector is a string requesting that only messages with property values (see "JMS Message Structure" on page 37) that match the string are delivered to a particular consumer. For example, the selector `NumberOfOrders >1` delivers only the messages with a `NumberOfOrders` property value of `2` or more.

Registering consumers with selectors lowers performance (as compared to using multiple destinations) because additional processing is required to handle each message. When a selector is used, it must be parsed so that it can be matched against future messages. Additionally, the message properties of each message must be retrieved and compared against the selector as each message is routed. However, using selectors provides more flexibility in a messaging application.

Message Size

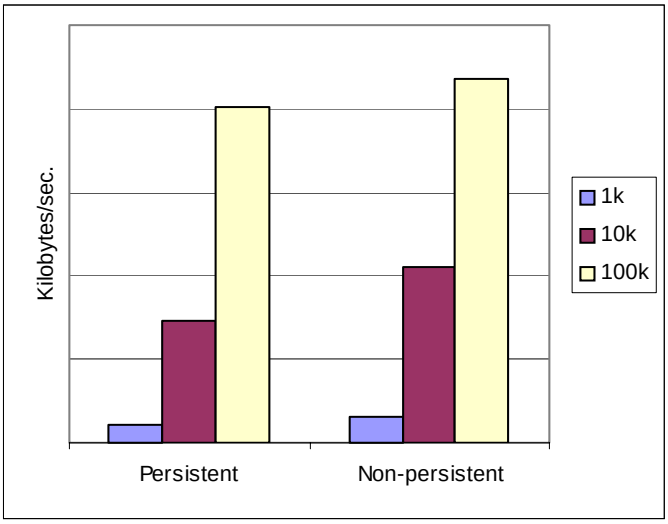
Message size affects performance because more data must be passed from producing client to broker and from broker to consuming client, and because for persistent messages a larger message must be stored.

However, by batching smaller messages into a single message, the routing and processing of individual messages can be minimized, providing an overall performance gain. In this case, information about the state of individual messages is lost.

Figure 9-4 compares throughput in kilobytes per second for 1k, 10k, and 100k-sized messages in two cases: persistent and non-persistent messages. All cases send messages are to a queue destination and use `AUTO_ACKNOWLEDGE` acknowledgement mode.

Figure 9-4 shows that in both cases there is less overhead in delivering larger messages compared to smaller messages. You can also see that the almost 50% performance gain of non-persistent messages over persistent messages shown for 1k and 10k-sized messages is not maintained for 100k-sized messages, probably because network bandwidth has become the bottleneck in message throughput for that case.

Figure 9-4 Performance Impact of a Message Size



Message Body Type

JMS supports five message body types, shown below roughly in the order of complexity:

- BytesMessage: Contains a set of bytes in a format determined by the application
- TextMessage: Is a simple java.lang.String
- StreamMessage: Contains a stream of Java primitive values
- MapMessage: Contains a set of name-and-value pairs
- ObjectMessage: Contains a Java serialized object

While, in general, the message type is dictated by the needs of an application, the more complicated types (MapMessage and ObjectMessage) carry a performance cost -- the expense of serializing and deserializing the data. The performance cost depends on how simple or how complicated the data is.

Message Service Factors that Impact Performance

The performance of a messaging application is affected not only by application design, but also by the message service performing the routing and delivery of messages.

The following sections discuss various message service factors that can affect performance. Understanding the impact of these factors is key to sizing a message service and diagnosing and resolving performance bottlenecks that might arise in a deployed application.

The most important factors affecting performance in an MQ message service are the following:

- Hardware
- Operating System
- Java Virtual Machine (JVM)
- Connections
- Broker Limits and Behaviors
- Message Server Architecture
- Data Store Performance
- Client Runtime Configuration

The sections below describe the impact of each of these factors on messaging performance.

Hardware

For both the MQ message server and client applications, cpu processing speed and available memory are primary determinants of message service performance. Many software limitations can be eliminated by increasing processing power, while adding memory can increase both processing speed and capacity. However, it is generally expensive to overcome bottlenecks simply by upgrading your hardware.

Operating System

Because of the efficiencies of different operating systems, performance can vary, even assuming the same hardware platform. For example, the thread model employed by the operating system can have an important impact on the number of concurrent connections a message server can support. In general, all hardware being equal, Solaris is generally faster than Linux, which is generally faster than Windows.

Java Virtual Machine (JVM)

The MQ message server is a Java process that runs in and is supported by the host JVM. As a result, JVM processing is an important determinant of how fast and efficiently a message server can route and deliver messages.

In particular, the JVM's management of memory resources can be critical. Sufficient memory has to be allocated to the JVM to accommodate increasing memory loads. In addition, the JVM periodically reclaims unused memory, and this memory reclamation can delay message processing. The larger the JVM memory heap, the longer the potential delay that might be experienced during memory reclamation.

Connections

The number and speed of connections between client and broker can affect the number of messages that a message server can handle as well as the speed of message delivery.

Message Server Connection Limits

All access to the message server is by way of connections. Any limit on the number of concurrent connections can affect the number of producing or consuming clients that can concurrently use the message server.

The number of connections to a message server is generally limited by the number of threads available. MQ uses a thread pool manager, which you can configure to support either a dedicated thread model or a shared thread model (see "Thread Pool Manager" on page 56). The dedicated thread model is very fast because each connection has dedicated threads, however the number of connections is limited by the number of threads available (one input thread and one output thread for each connection). The shared thread model places no limit on the number of connections, however there is significant overhead and throughput delays in sharing threads among a number of connections, especially when those connections are busy.

Transport Protocols

MQ software allows clients to communicate with the message server using various low-level transport protocols. MQ supports the connection services (and corresponding protocols) shown in "Connection Services Support" on page 55. The choice of protocols is based on application requirements (encrypted, accessible through a firewall), but the choice impacts overall performance.

Figure 9-5 Transport Protocol Speeds



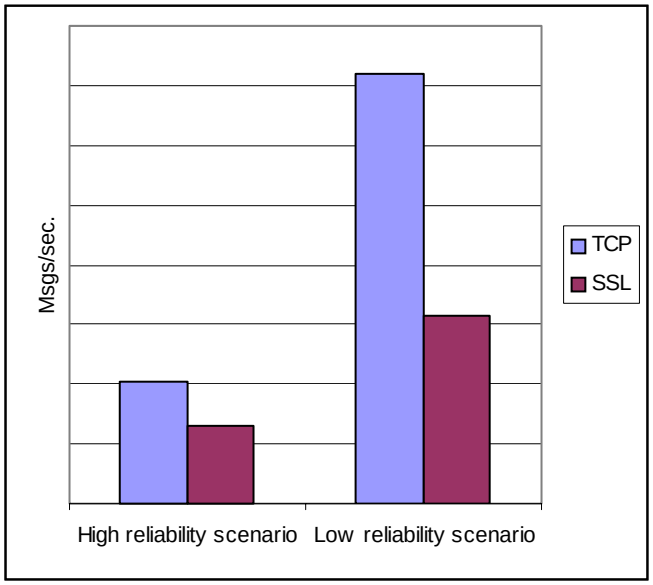
Figure 9-5 reflects the performance characteristics of the various protocol technologies:

- TCP provides the fastest method to communicate with the broker.
- SSL is 50 to 70 percent slower than TCP when it comes to sending and receiving messages (50 percent for persistent messages, closer to 70 percent for non-persistent messages). Additionally, establishing the initial connection is slower with ssl (it might take several seconds) because the client and broker (or Web Server in the case of HTTPS) need to establish a private key to be used when encrypting the data for transmission. The performance drop is caused by the additional processing required to encrypt and decrypt each low-level TCP packet.

Figure 9-6 compares throughput for TCP and SSL for two cases: a high reliability scenario (1k persistent messages sent to topic destinations with durable subscriptions and using `AUTO_ACKNOWLEDGE` acknowledgement mode) and a high performance scenario (1k non-persistent messages sent to topic destinations without durable subscriptions and using `DUPS_OK_ACKNOWLEDGE` acknowledgement mode).

Figure 9-6 shows that protocol has less impact in the high reliability case. This is probably because the persistence overhead required in the high reliability case is a more important factor in limiting throughput than the protocol speed.

Figure 9-6 Performance Impact of Transport Protocol



- HTTP is slower than either the TCP or SSL. It uses a servlet that runs on a Web server as a proxy between the client and the broker. Performance overhead is involved in encapsulating packets in HTTP requests and in the requirement that messages go through two hops--client to servlet, servlet to broker--to reach the broker.
- HTTPS is slower than HTTP because of the additional overhead required to encrypt the packet between client and servlet and between servlet and broker.

Message Server Architecture

An MQ message server can be implemented as a single broker or as multiple interconnected broker instances—a broker cluster.

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

In general, this scaling works best if clients are evenly distributed across the cluster, especially message producing clients. Because of the overhead involved in delivering messages between the brokers in a cluster, clusters with limited numbers of connections or limited message delivery rates, might exhibit lower performance than a single broker.

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

For more information on clusters, see "Multi-Broker Clusters (Enterprise Edition)" on page 81 and "Working With Clusters (Enterprise Edition)" on page 140.

Broker Limits and Behaviors

The message throughput that a message server might be required to handle is a function of the use patterns of the messaging applications the message server supports. However, the message server is limited in resources: memory, cpu cycles, and so forth. As a result, it would be possible for a message server to become overwhelmed to the point where it becomes unresponsive or unstable.

The MQ message server has mechanisms built in for managing memory resources and preventing the broker from running out of memory. These mechanisms include configurable limits on the number of messages or message bytes that can be held by a broker or its individual destinations, and a set of behaviors that can be instituted when destination limits are reached (see "Managing Memory Resources and Message Flow" on page 61.

With careful monitoring and tuning, these configurable mechanisms can be used to balance the inflow and outflow of messages so that system overload cannot occur. While these mechanisms consume overhead and can limit message throughput, they nevertheless maintain operational integrity.

Data Store Performance

MQ supports both built-in and plugged-in persistence (see "Persistence Manager" on page 63). Built-in persistence is a file-based data store. Plugged-in persistence uses a Java Database Connectivity (JDBC™) interface and requires a JDBC-compliant data store.

The built-in persistence is significantly faster than plugged-in persistence; however, a JDBC-compliant database system might provide the redundancy, security, and administrative features needed for an application.

In the case of built-in persistence, you can maximize reliability by specifying that persistence operations synchronize the in-memory state with the data store. This helps eliminate data loss due to system crashes, but at the expense of performance.

Client Runtime Configuration

The MQ client runtime provides client applications with an interface to the MQ message service. It supports all the operations needed for clients to send messages to destinations and to receive messages from such destinations. The client runtime is configurable (by setting connection factory attribute values), allowing you to set properties and behaviors that can generally improve performance and message throughput.

For example, the MQ client runtime supports the following configurable behaviors:

- Connection flow metering (`imqConnectionFlowCount`), which helps you prevent congestion due to the flow of both JMS messages and MQ control messages across the same connection.
- Connection flow limits (`imqConnectionFlowLimit`), which helps you avoid client resource limitations by limiting the number of messages that can be delivered over a connection to the client runtime, waiting to be consumed.
- Consumer flow limits (`imqConsumerFlowLimit`), which helps improve load balancing among consumers in multi-consumer queue delivery situations (so no one consumer can be sent a disproportionate number of messages) and which helps prevent any one consumer on a connection from overwhelming other consumers on the connection. This property limits the number of messages per consumer that can be delivered over a connection to the client runtime, waiting to be consumed. This property can also be configured as a queue destination property (`consumerFlowLimit`).

For more information on these behaviors and the attributes used to configure them, see "Client Runtime Message Flow Adjustments" on page 286.

Monitoring an MQ Message Server

An MQ message server can be configured to provide metrics information that you can then use to monitor the performance of the message server. This section describes the various tools you can use to monitor an MQ message server and the metrics data that can be obtained using these tools.

For information on how to use metrics data to troubleshoot performance problems or to analyze and tune message server performance, see "Troubleshooting Performance Problems" on page 262.

Monitoring Tools

You can obtain metrics information using the following tools:

- MQ Command Utility (`imqcmd`)
- MQ Broker Log Files
- MQ Message-Based Monitoring API

The following sections describe how to use each of these tools to obtain metrics information. For a comparison of the different tools, see "Choosing the Right Monitoring Tool" on page 253.

MQ Command Utility (`imqcmd`)

The Command utility (imqcmd) is MQ's basic command line administration tool. It allows you to manage the broker and its connection services, as well as application-specific resources such as physical destinations, durable subscriptions, and transactions. The `imqcmd` command is documented in Chapter 6, "Broker and Application Management."

One of the capabilities of the `imqcmd` command is its ability to obtain metrics information for the broker as a whole, for individual connection services, and for individual destinations. To obtain metrics data, you generally use the `metrics` subcommand of `imqcmd`. Metrics data is written at an interval you specify, or the number of times you specify, to the console screen.

You can also use the `query` subcommand (see "imqcmd query" on page 248) to obtain a more limited subset of metrics data.

imqcmd metrics

The syntax and options of `imqcmd metrics` are shown in Table 9-2 and Table 9-3, respectively.

Table 9-2 `imqcmd metrics` Subcommand Syntax

Subcommand Syntax	Metrics Data Provided

`metrics bkr`
`[-b` *hostName:port*`]`
`[-m` *metricType*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
`[-u` *userName*
`[-p` *pasword*
or
	Displays broker metrics for the default broker or a broker at the specified host and port.

`metrics svc -n` *serviceName*
`[-b` *hostName:port*`]`
`[-m` *metricType*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
`[-u` *userName*
`[-p` *pasword*
or
	Displays metrics for the specified service on the default broker or on a broker at the specified host and port.

`metrics dst -t` *destType*
`-n` *destName*
`[-b` *hostName:port*`]`
`[-m` *metricType*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
`[-u` *userName*
`[-p` *pasword*
	Displays metrics information for the destination of the specified type and name.

Table 9-3 `imqcmd metrics` Subcommand Options

Subcommand Options	Description
`-b` *hostName:port*	Specifies the hostname and port of the broker for which metrics data is reported. The default is `localhost:7676`
`-int` *interval*	Specifies the interval (in seconds) at which to display the metrics. The default is 5 seconds.

Table 9-3 `imqcmd metrics` Subcommand Options

Subcommand Options	Description
`-m` *metricType*	Specifies the type of metric to display: **`ttl`** Displays metrics on messages and packets flowing into and out of the broker (default metric type) **`rts`** Displays metrics on rate of flow of messages and packets into and out of the broker (per second) **`cxn`** Displays connections, virtual memory heap, and threads (brokers and connection services only) **`con`** Displays consumer-related metrics (destinations only) **`dsk`** Displays disk usage metrics (destinations only)
`-msp` *numSamples*	Specifies the number of samples displayed in the output. The default is an unlimited number (infinite).
`-n` *destName*	Specifies the destination name of the destination (if any) for which metrics data is reported. There is no default.
`-n` *serviceName*	Specifies the connection service (if any) for which metrics data is reported. There is no default.
`-t` *destTyp*	Specifies the type (queue or topic) of the destination (if any) for which metrics data is reported. There is no default.
`-u` *userName*	Specifies your (the administrator's) name. If you omit this value, you will be prompted for it.
`-p` *password*	Specifies your (the administrator's) password. If you omit this value, you will be prompted for it.

Procedure: Using the metrics Subcommand to Display Metrics Data

This section describes the procedure for using the `metrics` subcommand to report metrics information.

➤ **To Use the metrics Subcommand**

1. Start the broker for which metrics information is desired.

   See "Starting a Broker" on page 134.

2. Issue the appropriate `imqcmd metrics` subcommand and options as shown in Table 9-2 and Table 9-3.

Metrics Outputs: imqcmd metrics

This section shows example metrics subcommand outputs for broker-wide, connection service, and destination metrics.

Broker-wide metrics. To get the rate of message and packet flow into and out of the broker at 10 second intervals, use the `metrics bkr` subcommand:

```
imqcmd metrics bkr -m rts -int 10 -u admin -p admin
```

This command produces output similar to the following (see data descriptions in Table 9-8 on page 256):

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

Connection service metrics. To get cumulative totals for messages and packets handled by the jms connection service, use the `metrics svc` subcommand:

```
imqcmd metrics svc -n jms -m ttl -u admin -p admin
```

This command produces output similar to the following (see data descriptions in Table 9-9 on page 258):

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

Destination metrics. To get metrics information about a destination, use the `metrics dst` subcommand:

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

This command produces output similar to the following (see data descriptions in Table 9-10 on page 260):

```
--------------------------------------------------------------------------------
  Msgs      Msg Bytes         Msg Count          Total Msg Bytes (k)     Largest
In   Out    In     Out    Current  Peak  Avg  Current  Peak     Avg    Msg (k)
--------------------------------------------------------------------------------
200  200  147200  147200     0     200    0      0      143      71        0
300  200  220800  147200    100    200   10     71      143      64        0
300  300  220800  220800     0     200    0      0      143      59        0
```

To get information about a destination's consumers, use the following `metrics dst` subcommand:

```
imqcmd metrics dst -t q -n SimpleQueue -m con -u admin -p admin
```

This command produces output similar to the following (see data descriptions in Table 9-10 on page 260):

```
------------------------------------------------------------------
   Active Consumers        Backup Consumers         Msg Count
Current  Peak    Avg    Current  Peak    Avg    Current  Peak  Avg
------------------------------------------------------------------
   1       1      0        0       0      0       944    1000  525
```

imqcmd query

The syntax and options of `imqcmd query` are shown in Table 9-4 along with a description of the metrics data provided by the command.

Table 9-4 `imqcmd query` Subcommand Syntax

Subcommand Syntax	Metrics Data Provided

`query bkr`
`[-b` *hostName:port*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
or
	Information on the current number of messages and message bytes stored in broker memory and persistent store (see "Displaying Broker Information" on page 159)

Table 9-4 `imqcmd query` Subcommand Syntax

Subcommand Syntax	Metrics Data Provided

`metrics svc -n` *serviceName*
`[-b` *hostName:port*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
or
	Information on the current number of allocated threads and number of connections for a specified connection service (see "Displaying Connection Service Information" on page 165)

`metrics dst -t` *destType*
`-n` *destName*
`[-b` *hostName:port*`]`
`[-int` *interval*`]`
`[-msp` *numSamples*`]`
	Information on the current number of producers, active and backup consumers, and messages and message bytes stored in memory and persistent store for a specified destination (see "Displaying Destination Information" on page 173)

NOTE Because of the limited metrics data provided by `imqcmd query`, this tool is not represented in the tables presented in the section, "Description of Metrics Data,"on page 255.

MQ Broker Log Files

The MQ logger takes information generated by broker code, a debugger, and a metrics generator and writes that information to a number of output channels: to standard output (the console), to a log file, and, on Solaris™ platforms, to the `syslog` daemon process. The logger is describe in "Logger" on page 71.

You can specify the type of information gathered by the logger as well as the type written to each of the output channels. In particular, you can specify that you want metrics information written out to a log file.

Procedure: Using Broker Log Files to Report Metrics Data

This section describes the procedure for using broker log files to report metrics information. For general information on configuring the logger, see "Logging" on page 147.

➤ **To Use Log Files to Report Metrics Information**

1. Configure the broker's metrics generation capability:

   a. Confirm `imq.metrics.enabled=true`

      Generation of metrics for logging is turned on by default.

b. Set the metrics generation interval to a convenient number of seconds.

`imq.metrics.interval=`*interval*

This value can be set in the `config.properties` file or using the `-metrics` *interval* command line option when starting up the broker.

2. Confirm that the logger gathers metrics information:

`imq.log.level=INFO`

This is the default value. This value can be set in the `config.properties` file or using the `-loglevel` *level* command line option when starting up the broker.

3. Confirm that the logger is set to write metrics information to the log file:

`imq.log.file.output=INFO`

This is the default value. It can be set in the `config.properties` file.

4. Start up the broker.

Metrics Outputs: Log File

The following shows sample broker metrics output to the log file (see the description of metrics data in Table 9-7 and Table 9-8 on page 256):

```
[21/Jul/2003:11:21:18 PDT]
Connections: 0    JVM Heap: 8323072 bytes (7226576 free) Threads: 0 (14-1010)
      In: 0 msgs (0bytes) 0 pkts (0 bytes)
     Out: 0 msgs (0bytes) 0 pkts (0 bytes)
 Rate In: 0 msgs/sec (0 bytes/sec) 0 pkts/sec (0 bytes/sec)
Rate Out: 0 msgs/sec (0 bytes/sec) 0 pkts/sec (0 bytes/sec)
```

MQ Message-Based Monitoring API

MQ provides a metrics monitoring capability by which the broker can write metrics data into JMS messages, which it then sends to one of a number of metrics topic destinations, depending on the type of metrics information contained in the message.

You can access this metrics information by writing a client application that subscribes to the metrics topic destinations, consumes the messages in these destinations, and processes the metrics information contained in the messages. The general scheme is described in "Metrics Message Producer (Enterprise Edition)" on page 72.

There are five metrics topic destinations, whose names are shown in Table 9-5, along with the type of metrics messages delivered to each destination.

Table 9-5 Metrics Topic Destinations

Topic Name	Type of Metrics Messages
mq.metrics.broker	Broker metrics
mq.metrics.jvm	Java Virtual Machine metrics
mq.metrics.destination_list	List of destinations and their types
mq.metrics.destination.queue.*monitoredDestinationName*	Destination metrics for queue of specified name
mq.metrics.destination.topic.*monitoredDestinationName*	Destination metrics for topic of specified name

Procedure: Setting Up Message-Based Monitoring

This section describes the procedure for using the MQ message-based monitoring capability to gather metrics information. The procedure includes both client development and administration tasks.

➤ **To Set Up Message-based Monitoring**

1. Write a metrics monitoring client.

   See the MQ *Java Client Developer's Guide* for instructions on programming clients that subscribe to metrics topic destinations, consume metrics messages, and extract the metrics data from these messages.

2. Configure the broker's Metrics Message Producer by setting broker property values in the `config.properties` file:

   a. Enable metrics message production.

      Set `imq.metrics.topic.enabled=true`

      The default value is `true`.

   b. Set the interval (in seconds) at which metrics messages are generated.

      Set `imq.metrics.topic.interval=`*interval*

      The default is 60 seconds.

c. Specify if you wish metrics messages to be persistent (that is, whether they will survive a broker failure).

Set `imq.metrics.topic.persist`

The default is `false`.

d. Specify how long you want metrics messages to remain in their respective destinations before being deleted.

Set `imq.metrics.topic.timetolive`

The default value is 300 seconds

3. Set any access control you desire on metrics topic destinations.

See the discussion in "Security and Access Considerations," below.

4. Start up your metrics monitoring client.

When consumers subscribe to a metrics topic, the metrics topic destination will automatically be created. Once a metrics topic has been created, the broker's metrics message producer will begin sending metrics messages to the metrics topic.

Security and Access Considerations

There are two reasons to restrict access to metrics topic destinations:

- Metrics data might include sensitive information about a broker and its resources
- Excessive numbers of subscriptions to metrics topic destinations might increase broker overhead and negatively impact performance

Because of these considerations, it is advisable to restrict access to metrics topic destinations.

Monitoring clients are subject to the same authentication and authorization control as any other client. Only users maintained in the MQ user repository are allowed to connect to the broker.

You can provide additional protections by restricting access to specific metrics topic destinations through an access control properties file, as described in "Authorizing Users: the Access Control Properties File" on page 212.

For example, the following entries in an `accesscontrol.properties` file will deny access to the mq.metrics.broker metrics topic to everyone except user1 and user 2.

```
topic.mq.metrics.broker.consume.deny.user=*
topic.mq.metrics.broker.consume.allow.user=user1,user2
```

The following entries will only allow users user3 to monitor topic t1.

```
topic.mq.metrics.destination.topic.t1.consume.deny.user=*
topic.mq.metrics.destination.topic.t1.consume.allow.user=user3
```

Depending on the sensitivity of metrics data, you can also connect your metrics monitoring client to a broker using an encrypted connection. For information on using encrypted connections, see "Encryption: Working With an SSL-based Service (Enterprise Edition)" on page 218.

Metrics Outputs: Metrics Messages

The metrics data outputs you get using the message-based monitoring API is a function of the metrics monitoring client you write. You are limited only by the data provided by the metrics generator in the broker. For a complete list of this data, see "Description of Metrics Data" on page 255.

Choosing the Right Monitoring Tool

Each of the monitoring tools discussed in the previous sections has its advantages and disadvantages.

Using the `imqcmd metrics` command, for example, lets you quickly sample information tailored to your needs when you want it, but makes it somewhat difficult to look at historical information, or to manipulate the data programmatically.

The log files, on the other hand, provide a long-term record of metrics data, however the information in the log file is difficult to parse for meaningful information.

The message-based monitoring API lets you easily extract the information you need, process it, manipulate or format the data programmatically, present graphs or send alerts; however, you have to write a custom application to capture and analyze the data.

In addition, each of these tools gathers a somewhat different subset of the metrics information generated by the broker. For information on which metrics data is gathered by which monitoring tool, see "Description of Metrics Data" on page 255.

Table 9-6 compares the different tools by showing the pros and cons of each.

Table 9-6 Pros and Cons of Metrics Monitoring Tools

Metrics Monitoring Tool	Pros	Cons
`imqcmd metrics`	Remote monitoring Convenient for spot checking Reporting interval set in command option; can be changed on the fly Easy to select specific data of interest Data presented in easy tabular format	No single command gets all data Difficult to analyze data programmatically Doesn't create historical record Difficult to see historical trends
Log files	Regular sampling Creates a historical record	Need to configure broker properties; must shut down and restart broker to take effect Local monitoring only Data format very difficult to read or parse; no parsing tools Reporting interval cannot be changed on the fly; the same for all metrics data Does not provide flexibility in selection of data Broker metrics only; destination and connection service metrics not included Possible performance hit if interval set too short
Message-based monitoring API	Remote monitoring Easy to select specific data of interest Data can be analyzed electronically and presented in any format	Need to configure broker properties; must shut down and restart broker to take effect You need to write your own metrics monitoring client Reporting interval cannot be changed on the fly; the same for all metrics data

Description of Metrics Data

The metrics information reported by a broker can be grouped into the following categories:

- **Java Virtual Machine (JVM) metrics**. Information about the JVM heap size.
- **Broker-wide metrics**. Information about messages stored in a broker and about message flows into and out of a broker, both in terms of numbers of messages and numbers of bytes (in absolute terms as well as rates). This category also includes information about memory usage.
- **Connection Service metrics**. Information about connections and connection thread resources, as well as information about message flows for a particular connection service.
- **Destination metrics**. Information about message flows into and out of a particular destination, information about a destination's consumers, and information about memory and disk space usage.

The following sections present the metrics data available in each of these categories. For information on the monitoring tools referred to in the following tables, see "Monitoring Tools" on page 244.

JVM Metrics

Table 9-7 lists and describes the metrics data the broker generates for the broker process JVM heap and shows which of the data can be obtained using the different metrics monitoring tools.

Table 9-7 JVM Metrics

Metric Quantity	Description	imqcmd `metrics bkr` (*metricType*)	Log File	Metrics Message (*metrics topic*)[2]
JVM heap: free memory	The amount of free memory available for use in the JVM heap	Yes (`cxn`)	Yes	Yes (`…jvm`)
JVM heap: total memory	The current JVM heap size	Yes (`cxn`)	Yes	Yes (`…jvm`)
JVM heap: max memory	The maximum to which the JVM heap size can grow.	No	Yes[1]	Yes (`…jvm`)

1. Shown only at broker startup.
2. For metrics topic destination names, see Table 9-5 on page 251.

Broker-wide Metrics

Table 9-8 lists and describes the data the broker reports regarding broker-wide metrics information. It also shows which of the data can be obtained using the different metrics monitoring tools.

Table 9-8 Broker-wide Metrics

Metric Quantity	Description	imqcmd `metrics bkr` (*metricType*)	Log File	Metrics Message (*metrics topic*)[1]
	Connection Data			
Num connections	Number of currently open connections to the broker	Yes (`cxn`)	Yes	Yes (…broker)
Num threads	Number of threads currently in use	Yes (`cxn`)	Yes	No
Min threads	Number of threads, which once reached, are maintained in the thread pool for use by connection services	Yes (`cxn`)	Yes	No
Max threads	Number of threads, beyond which no new threads are added to the thread pool for use by connection services	Yes (`cxn`)	Yes	No
	Stored Messages Data			
Num messages	Number of JMS messages currently stored in broker memory and persistent store	No Use `query bkr`	No	Yes (…broker)
Total message bytes	Number of JMS messages bytes currently stored in broker memory and persistent store	No Use `query bkr`	No	Yes (…broker)
	Message Flow Data			
Num messages in	Number of JMS messages that have flowed into the broker since it was last started	Yes (`ttl`)	Yes	Yes (…broker)
Message bytes in	Number of JMS message bytes that have flowed into the broker since it was last started	Yes (`ttl`)	Yes	Yes (…broker)
Num packets in	Number of packets that have flowed into the broker since it was last started; includes both JMS messages and control messages	Yes (`ttl`)	Yes	Yes (…broker)
Packet bytes in	Number of packet bytes that have flowed into the broker since it was last started; includes both JMS messages and control messages	Yes (`ttl`)	Yes	Yes (…broker)

Table 9-8 Broker-wide Metrics *(Continued)*

Metric Quantity	Description	imqcmd `metrics bkr` (*metricType*)	Log File	Metrics Message (*metrics topic*)[1]
Num messages out	Number of JMS messages that have flowed out of the broker since it was last started.	Yes (`ttl`)	Yes	Yes (…broker)
Message bytes out	Number of JMS message bytes that have flowed out of the broker since it was last started	Yes (`ttl`)	Yes	Yes (…broker)
Num packets out	Number of packets that have flowed out of the broker since it was last started; includes both JMS messages and control messages	Yes (`ttl`)	Yes	Yes (…broker)
Packet bytes out	Number of packet bytes that have flowed out of the broker since it was last started; includes both JMS messages and control messages	Yes (`ttl`)	Yes	Yes (…broker)
Rate messages in	Current rate of flow of JMS messages into the broker	Yes (`rts`)	Yes	No
Rate message bytes in	Current rate of flow of JMS message bytes into the broker	Yes (`rts`)	Yes	No
Rate packets in	Current rate of flow of packets into the broker; includes both JMS messages and control messages	Yes (`rts`)	Yes	No
Rate packet bytes in	Current rate of flow of packet bytes into the broker; includes both JMS messages and control messages	Yes (`rts`)	Yes	No
Rate messages out	Current rate of flow of JMS messages out of the broker	Yes (`rts`)	Yes	No
Rate message bytes out	Current rate of flow of JMS message bytes out of the broker	Yes (`rts`)	Yes	No
Rate packets out	Current rate of flow of packets out of the broker; includes both JMS messages and control messages	Yes (`rts`)	Yes	No
Rate packet bytes out	Current rate of flow of packet bytes out of the broker; includes both JMS messages and control messages	Yes (`rts`)	Yes	No
	Destinations Data			
Num destinations	Number of physical destination in the broker	No	No	Yes (…broker)

1. For metrics topic destination names, see Table 9-5 on page 251.

Connection Service Metrics

Table 9-9 lists and describes the metrics data the broker reports for individual connection services. It also shows which of the data can be obtained using the different metrics monitoring tools.

Table 9-9 Connection Service Metrics

Metric Quantity	Description	imqcmd `metrics svc` (*metricType*)	Log File	Metrics Message (*metrics topic*)
	Connection Data			
Num connections	Number of currently open connections	Yes (`cxn`) Also `query svc`	No	No
Num threads	Number of threads currently in use, totaled across all connection services	Yes (`cxn`) Also `query svc`	No	No
Min threads	Number of threads, which once reached, are maintained in the thread pool for use by connection services, totaled across all connection services	Yes (`cxn`)	No	No
Max threads	Number of threads, beyond which no new threads are added to the thread pool for use by connection services, totaled across all connection services	Yes (`cxn`)	No	No
	Message Flow Data			
Num messages in	Number of JMS messages that have flowed into the connection service since the broker was last started	Yes (`ttl`)	No	No
Message bytes in	Number of JMS message bytes that have flowed into the connection service since the broker was last started	Yes (`ttl`)	No	No
Num packets in	Number of packets that have flowed into the connection service since the broker was last started; includes both JMS messages and control messages	Yes (`ttl`)	No	No
Packet bytes in	Number packet bytes that have flowed into the connection service since the broker was last started; includes both JMS messages and control messages	Yes (`ttl`)	No	No
Num messages out	Number of JMS messages that have flowed out of the connection service since the broker was last started.	Yes (`ttl`)	No	No

Table 9-9 Connection Service Metrics *(Continued)*

Metric Quantity	Description	imqcmd `metrics svc` (*metricType*)	Log File	Metrics Message (*metrics topic*)
Message bytes out	Number of JMS message bytes that have flowed out of the connection service since the broker was last started	Yes (`ttl`)	No	No
Num packets out	Number of packets that have flowed out of the connection service since the broker was last started; includes both JMS messages and control messages	Yes (`ttl`)	No	No
Packet bytes out	Number packet bytes that have flowed out of the connection service since the broker was last started; includes both JMS messages and control messages	Yes (`ttl`)	No	No
Rate messages in	Current rate of flow of JMS messages into the broker through the connection service.	Yes (`rts`)	No	No
Rate message bytes in	Current rate of flow of JMS message bytes into the connection service	Yes (`rts`)	No	No
Rate packets in	Current rate of flow of packets into the connection service; includes both JMS messages and control messages	Yes (`rts`)	No	No
Rate packet bytes in	Current rate of flow of packet bytes into the connection service; includes both JMS messages and control messages	Yes (`rts`)	No	No
Rate messages out	Current rate of flow of JMS messages out of the connection service	Yes (`rts`)	No	No
Rate message bytes out	Current rate of flow of JMS message bytes out of the connection service	Yes (`rts`)	No	No
Rate packets out	Current rate of flow of packets out of the connection service; includes both JMS messages and control messages	Yes (`rts`)	No	No
Rate packet bytes out	Current rate of flow of packet bytes out of the connection service; includes both JMS messages and control messages	Yes (`rts`)	No	No

Destination Metrics

Table 9-9 lists and describes the metrics data the broker reports for individual destinations. It also shows which of the data can be obtained using the different metrics monitoring tools.

Table 9-10 Destination Metrics

Metric Quantity	Description	imqcmd `metrics dst` (*metricType*)	Log File	Metrics Message (*metrics topic*)[1]
	Consumer Data			
Num active consumers	Current number of active consumers	Yes (`con`)	No	Yes (…*destName*)
Avg num active consumers	Average number of active consumers since the broker was last started	Yes (`con`)	No	Yes (…*destName*)
Peak num active consumers	Peak number of active consumers since the broker was last started	Yes (`con`)	No	Yes (…*destName*)
Num backup consumers	Current number of backup consumers (applies only to queues)	Yes (`con`)	No	Yes (…*destName*)
Avg num backup consumers	Average number of backup consumers since the broker was last started (applies only to queues)	Yes (`con`)	No	Yes (…*destName*)
Peak num backup consumers	Peak number of backup consumers since the broker was last started (applies only to queues)	Yes (`con`)	No	Yes (…*destName*)
	Stored Messages Data			
Num messages	Number of JMS messages currently stored in destination memory and persistent store	Yes (`con`) (`ttl`) (`rts`) Also `query dst`	No	Yes (…*destName*)
Avg num messages	Average number of JMS messages stored in destination memory and persistent store since the broker was last started	Yes (`con`) (`ttl`) (`rts`)	No	Yes (…*destName*)
Peak num messages	Peak number of JMS messages stored in destination memory and persistent store since the broker was last started	Yes (`con`) (`ttl`) (`rts`)	No	Yes (…*destName*)

Table 9-10 Destination Metrics *(Continued)*

Metric Quantity	Description	imqcmd `metrics dst` (*metricType*)	Log File	Metrics Message (*metrics topic*)[1]
Total message bytes	Number of JMS message bytes currently stored in destination memory and persistent store	Yes (`ttl`) (`rts`) Also `query dst`	No	Yes (…*destName*)
Avg total message bytes	Average number of JMS message bytes stored in destination memory and persistent store since the broker was last started	Yes (`ttl`) (`rts`)	No	Yes (…*destName*)
Peak total message bytes	Peak number of JMS message bytes stored in destination memory and persistent store since the broker was last started	Yes (`ttl`) (`rts`)	No	Yes (…*destName*)
Peak message bytes	Peak number of JMS message bytes in a single message received by the destination since the broker was last started	Yes (`ttl`) (`rts`)	No	Yes (…*destName*)
	Message Flow Data			
Num messages in	Number of JMS messages that have flowed into this destination since the broker was last started	Yes (`ttl`)	No	Yes (…*destName*)
Msg bytes in	Number of JMS message bytes that have flowed into this destination since the broker was last started	Yes (`ttl`)	No	Yes (…*destName*)
Num messages out	Number of JMS messages that have flowed out of this destination since the broker was last started	Yes (`ttl`)	No	Yes (…*destName*)
Msg bytes out	Number of JMS message bytes that have flowed out of this destination since the broker was last started	Yes (`ttl`)	No	Yes (…*destName*)
Rate num messages in	Current rate of flow of JMS messages into the destination	Yes (`rts`)	No	No
Rate num messages out	Current rate of flow of JMS messages out of the destination	Yes (`rts`)	No	No
Rate msg bytes in	Current rate of flow of JMS message bytes into the destination	Yes (`rts`)	No	No
Rate Msg bytes out	Current rate of flow of JMS message bytes out of the destination	Yes (`rts`)	No	No

Table 9-10 Destination Metrics *(Continued)*

Metric Quantity	Description	imqcmd `metrics dst` (*metricType*)	Log File	Metrics Message (*metrics topic*)[1]
	Disk Utilization Data			
Disk reserved	Disk space (in bytes) used by all message records (active and free) in the destination file-based store	Yes (`dsk`)	No	Yes (…*destName*)
Disk used	Disk space (in bytes) used by active message records in destination file-based store	Yes (`dsk`)	No	Yes (…*destName*)
Disk utilization ratio	Quotient of used disk space over reserved disk space. The higher the ratio, the more the disk space is being used to hold active messages	Yes (`dsk`)	No	Yes (…*destName*)

1. For metrics topic destination names, see Table 9-5 on page 251.

Troubleshooting Performance Problems

There are a number of performance problems that can occur in using an MQ message service to support a messaging application. These problems include the following:

- Problem: Clients Can't Establish A Connection
- Problem: Connection Throughput is Too Slow
- Problem: Client Can't Create a Message Producer
- Problem: Message Production Is Delayed or Slowed
- Problem: Messages Backlogged in Message Server
- Problem: Message Server Throughput Is Sporadic
- Problem: Messages Not Reaching Consumers

Each of these problems is discussed below along with possible causes and solutions.

Problem: Clients Can't Establish A Connection

Symptoms:

- Client cannot make a new connection.
- Client cannot auto-reconnect on failed connection.

Possible Causes:

- Client applications are not closing connections, causing the number of connections to exceed resource limitations.

  To confirm this cause of the problem:

  List all connections to a broker:

  ```
  imqcmd list cxn
  ```

  The output will list all connections and the host from which each connection has been made, revealing an unusual number of open connections for specific clients.

  To resolve the problem:

  Rewrite the offending clients to close unused connections.

- Broker is not running or there is a network connectivity problem.

  To confirm this cause of the problem:

  - Telnet to the broker's primary port (for example, the default of 7676) and verify that the broker responds with Port Mapper output.
  - Verify that the broker process is running on the host.

  To resolve the problem:

  - Start up the broker.
  - Fix the network connectivity problem.

- Connection service is inactive or paused.

  To confirm this cause of the problem:

  Check the status of all connection services:

  ```
  imqcmd list svc
  ```

If the status of a connection service is shown as `unknown` or `paused`, then clients will not be able to establish a connection using that service.

To resolve the problem:

- If the status of a connection service is shown as `unknown`, then it is missing from the active service list (`imq.service.active`). In the case of SSL-based services, the service might also be improperly configured, causing the broker to make the following entry in the broker log: `ERROR [B3009]: Unable to start service ssljms: [B4001]: Unable to open protocol tls for ssljms service...` followed by an explanation of the underlying cause of the exception.

  To properly configure SSL services, see "Setting Up an SSL-based Service Over TCP/IP" on page 218.

- If the status of a connection service is shown as `paused`, then resume the service (see "Pausing and Resuming a Connection Service" on page 166).

- Too few threads available for the number of connections required.

  To confirm this cause of the problem:

  Check for the following entry in the broker log: `WARNING [B3004]: No threads are available to process a new connection on service ... Closing the new connection.`

  Also check the number of connections on the connection service and the number of threads currently in use:

  `imqcmd query svc -n` *serviceName*
  or
  `imqcmd metrics svc -n` *serviceName* `-m cxn`

  Each connection requires two threads: one for incoming messages and one for outgoing messages (see "Thread Pool Manager" on page 56).

  To resolve the problem:

  - If you are using a dedicated thread pool model (`imq.`*service_name*`.threadpool_model=dedicated`), the maximum number of connections is half the maximum number of threads in the thread pool. Therefore, to increase the number of connections, increase the size of the thread pool (`imq.`*service_name*`.max_threads`) or switch to the shared thread pool model.

- If you are using a shared thread pool model (`imq.`*service_name*`.threadpool_model=shared`), the maximum number of connections is half the product of the following two properties: the connection Monitor limit (`imq.`*service_name*`.connectionMonitor_limit`) and the maximum number of threads (`imq.`*service_name*`.max_threads`). Therefore, to increase the number of connections, increase the size of the thread pool or increase the connection monitor limit.
- Ultimately, the number of supportable connections (or the throughput on connections) will reach input/output limits. In such cases, use a multi-broker cluster (see "Working With Clusters (Enterprise Edition)" on page 140) to distribute connections among the broker instances within the cluster.

- Too few file descriptors for the number of connections required on the Solaris or Linux platform (see "OS-Defined File Descriptor Limitations" on page 338).

  To confirm this cause of the problem:

  Check for an entry in the broker log similar to the following: `Too many open files`.

  To resolve the problem:

  Increase the file descriptor limit, as described in the `ulimit` man page.

- TCP backlog limits the number of simultaneous new connection requests that can be established.

  The TCP backlog places a limit on the number of simultaneous connection requests that can be stored in the system backlog (`imq.portmapper.backlog`) before the Port Mapper rejects additional requests. (On Windows platforms there is a hard-coded backlog limit: 5 for Windows desktops and 200 for Windows servers.)

  The rejection of requests because of backlog limits is usually a transient phenomenon, due to an unusually high number of simultaneous connection requests.

  To confirm this cause of the problem:

  Check the broker log to see if some connection requests are being accepted while others at about the same time are being rejected. Rejected connection requests return a `java.net.ConnectException: Connection refused`.

  To resolve the problem:

  The following approaches can be used to resolve TCP backlog limitations:

- Program the client to retry the attempted connection after a short interval of time (this normally works because of the transient nature of this problem).
- Increase the value of `imq.portmapper.backlog`.
- Check that clients are not closing and then opening connections too often.

- Operating system limits the number of concurrent connections.

  The Windows operating system license places limits on the number of concurrent remote connections that are supported.

  To confirm this cause of the problem:

  Check that there are plenty of threads available for connections (using `imqcmd query svc`) and check the terms of your Windows license agreement. If you can make connections from a local client, but not from a remote client, then operating system limitations might be the cause of the problem.

  To resolve the problem:

  - Upgrade the Windows license to allow more connections.
  - Distribute connections among a number of broker instances by setting up a multi-broker cluster.

- Authentication or authorization of the user is failing.

  The authentication can be failing due to an incorrect password, because there is no entry for the user in the user repository, or because the user does not have access permissions for the connection service.

  To confirm this cause of the problem:

  Check entries in the broker log for the `Forbidden` error message. This will indicate an authentication error, but will not indicate the reason for it.

  - If you are using a file-based user repository, enter the following command:

    `imqusermgr list -i` *instanceName* `-u userName`

    If the output shows a user, then the wrong password was probably submitted. If the output shows an Error `[B3048]: User does not exist in the password file`, then there is no entry in the user repository.
  - If you are using an LDAP server user repository, use the appropriate tools to check if there is an entry for the user.
  - Check the access control properties file to see if there are restrictions on access to the connection service.

To resolve the problem:

- If there is no entry for the user in the user repository, then add the user to the user repository (see "Populating and Managing a User Repository" on page 207).
- If the wrong password was used, provide the correct password.
- If the access control properties are improperly set, edit the access control properties file to grant connection service permissions (see "Connection Access Control" on page 215).

Problem: Connection Throughput is Too Slow

Symptoms:

- Message throughput does not meet expectations.
- The number of supported connections to a broker is not limited as described in "Problem: Clients Can't Establish A Connection" on page 263, but rather by message input/output rates.

Possible Causes:

- Network connection or WAN is too slow.

  To confirm this cause of the problem:

  Ping the network to see how long it takes for the ping to return, and then consult a network administrator. Also you can send and receive messages using local clients and compare the delivery time with that of remote clients (which use a network link).

  To resolve the problem:

  If the connection is too slow, upgrade the network link.

- Connection service protocol is inherently slow compared to TCP. For example, SSL-based or HTTP-based protocols are slower than TCP (see Figure 9-5 on page 240).

  To confirm this cause of the problem:

  If you are using SSL-based or HTTP-based protocols, try using TCP and compare the delivery times.

  To resolve the problem:

Application requirements usually dictate the protocols being used, so there is little that you can do, other than to attempt to tune the protocol as described in ("Tuning Transport Protocols" on page 281).

- Connection service protocol is not optimally tuned.

  To confirm this cause of the problem:

  Try tuning the protocol and see if it makes a difference.

  To resolve the problem:

  Try tuning the protocol as described in ("Tuning Transport Protocols" on page 281).

- Messages are so large they consume too much bandwidth.

  To confirm this cause of the problem:

  Try running your benchmark with smaller-sized messages.

  To resolve the problem:

  - Compress message bodies using java.util.zip.
  - Use messages as notifications of data to be sent, but move the data using another protocol.

- What appears to be slow connection throughput is actually a bottleneck in some other step of the message delivery process.

  To confirm this cause of the problem:

  If none of the items above appear to be the cause of what appears to be slow connection throughput, consult Figure 9-1 on page 229 for other possible bottlenecks and check for symptoms associated with the following problems:

  - "Problem: Message Production Is Delayed or Slowed" on page 270
  - "Problem: Messages Backlogged in Message Server" on page 273
  - "Problem: Message Server Throughput Is Sporadic" on page 277

  To resolve the problem:

  Follow the problem resolution guidelines provided in the problem troubleshooting sections above.

Problem: Client Can't Create a Message Producer

Symptoms:

- A message producer cannot be created for a destination; the client receives an exception.

Possible Causes:

- A destination has been configured to allow only a limited number of producers.

  One of the ways of avoiding the accumulation of messages on a destination is to limit the number of producers (`maxNumProducers`) that can be supported by the destination.

  To confirm this cause of the problem:

  Check the destination (see "Displaying Destination Information" on page 173):

  ```
  imqcmd query dst
  ```

  The output will show the current number of producers and the value of `maxNumProducers`. If the two values are the same, then the number of producers has reached its configured limit. When a new producer is rejected by the broker, the broker returns a `ResourceAllocationException [C4088]: A JMS destination limit was reached` and makes the following entry in the broker log: `[B4183]: Producer can not be added to destination`.

  To resolve the problem:

  Increase the value of the `maxNumProducers` attribute (see "Updating Destination Attributes" on page 174).

- The user is not authorized to create a message producer due to settings in the access control properties file.

  To confirm this cause of the problem:

  When a new producer is rejected by the broker, the broker returns a `JMSSecurityException [C4076]: Client does not have permission to create producer on destination` and makes the following entries in the broker log: `[B2041]: Producer on destination denied` and `[B4051]: Forbidden guest`.

To resolve the problem:

Change the access control properties to allow the user to produce messages (see "Destination Access Control" on page 216).

Problem: Message Production Is Delayed or Slowed

Symptoms:

- When sending persistent messages, the `send()` method does not return and the client blocks.
- When sending a persistent message, client receives an exception.
- Producing client slows down.

Possible Causes:

- The message server is backlogged (messages are accumulating in broker memory) and has responded by slowing message producers.

  When the number of messages or number of message bytes in destination memory reaches configured limits, the broker attempts to conserve memory resources in accordance with the specified limit behavior. The following limit behaviors slow down message producers:

  - `FLOW_CONTROL`: the broker does not immediately acknowledge receipt of persistent messages (thereby blocking a producing client)
  - `REJECT_NEWEST`: the broker rejects new messages (and throws an exception for each rejected persistent message).

  Similarly, when the number of messages or number of message bytes in broker-wide memory (for all destinations) reaches configured limits, the broker will attempt to conserve memory resources by rejecting the newest messages.

  Also, when system memory limits are reached (because destination or broker-wide limits have not been set properly), the broker takes increasingly serious action to prevent memory overload, including throttling back message producers.

  For a discussion of these mechanisms, see "Managing Memory Resources and Message Flow" on page 61).

To confirm this cause of the problem:

When a message is rejected by the broker due to configured message limits, the broker returns a `JMSException [C4036]: A server error occurred` and makes entries in the broker log: `WARNING [B2011]: Storing of JMS message from IMQconn failed`, followed by a message indicating the limit that has been reached:

- If the message limit is on a destination, the broker makes an entry like the following: `[B4120]: Can not store message on destination` *destName* `because capacity of` *maxNumMsgs* `would be exceeded`.
- If the message limit is broker wide, the broker makes an entry like the following: `[B4024]: The Maximum Number of messages currrently in the system has been exceeded, rejecting message`.

More generally, you can check for message limit conditions before the rejections occur by querying destinations and the broker and inspecting their configured message limit settings, and by monitoring the number of messages or number of message bytes currently in a destination (or in the broker as a whole) using the appropriate `imqcmd` commands (see Table 9-10 on page 260 and Table 9-8 on page 256, respectively).

To resolve the problem:

There are a number of approaches to addressing the slowing of producers due to messages becoming backlogged:

- Modify the message limits on a destination (or broker-wide) being careful not to exceed memory resources. In general, you want to manage memory on a destination-by-destination level so that broker-wide message limits are never reached. For more information, see "Broker Adjustments" on page 285.
- Change the limit behaviors on a destination to not slow message production when message limits are reached, but rather to discard messages in memory. For example, you can specify the `REMOVE_OLDEST` and `REMOVE_LOW_PRIORITY` limit behaviors, which delete messages that accumulate in memory (see Table 6-10 on page 171).

- Broker cannot save a persistent message to the data store.

  If the broker cannot access a data store or write a persistent message to the data store, then the producing client is blocked. This condition can also occur if destination or broker-wide message limits are reached, as described above.

  To confirm this cause of the problem:

If the broker is unable to write to the data store, it makes one of the following entries in the broker log: `[B2011]: Storing of JMS message from connectionID failed`... or `[B4004]: Failed to persist message messageID`...

To resolve the problem:

- In the case of built-in persistence, try increasing the disk space of the file-based data store.
- In the case of a JDBC-compliant data store, check that plugged-in persistence is properly configured (see Appendix B, "Setting Up Plugged-in Persistence"). If so, consult your database administrator to troubleshoot other database problems.

- Broker acknowledgement timeout is too short.

  Due to slow connections or a lethargic message server (caused by high cpu utilization or scarce memory resources), a broker might require more time to acknowledge receipt of a persistent message than allowed by the value of the connection factory's `imqAckTimeout` attribute.

  To confirm this cause of the problem:

  If the `imqAckTimeout` value is exceeded, the broker returns a `JMSException [C4000]: Packet acknowledge failed.`

  To resolve the problem:

  Change the value of the `imqAckTimeout` connection factory attribute (see "Connection Factory Administered Object Attributes" on page 187).

- Producing client is encountering JVM limitations.

  To confirm this cause of the problem:

  - Check if the client application receives an Out Of Memory error.
  - Check the free memory available in the JVM heap using runtime methods such as `freeMemory()`, `MaxMemory()`, and `totalMemory()`.

  To resolve the problem:

  Adjust the JVM (see "Java Virtual Machine Adjustments" on page 281).

Problem: Messages Backlogged in Message Server

Symptoms:

- Number of messages or message bytes in broker (or in specific destinations) increases steadily over time.

  To see if messages are accumulating, check how the number of messages or message bytes in the broker changes over time and compare to configured limits. First check the configured limits:

  ```
  imqcmd query bkr
  ```

  (Note: the `imqcmd metrics bkr` subcommand does not display this information.)

  Then check for message accumulation in each destination:

  ```
  imqcmd query dst -t destType -n destName
  or
  imqcmd metrics dst -t destType -n destName -m ttl
  ```

  To see if messages have exceeded configured destination or broker-wide limits, check the broker log for the following entry: `WARNING [B2011]: Storing of JMS message from…failed`. This entry will be followed by another entry explaining the limit that has been exceeded.

- Message production is delayed or produced messages are rejected by the broker.

- Messages take an unusually long time to reach consumers.

Possible Causes:

- Client code defects: consumers are not acknowledging messages.

  Messages are held in a destination until they have been acknowledged by all consumers to which the messages have been sent. Hence, if a client is not acknowledging consumed messages, the messages accumulate in the destination without being deleted.

  For example, client code might have the following defects:

  - Consumers using `CLIENT_ACKNOWLEDGEMENT` or transacted session might not be calling `Session.acknowledge()` or `Session.commit()` on a regular basis.

- Consumers using `AUTO_ACKNOWLEDGE` sessions might be hanging for some reason.

To confirm this cause of the problem:

If a message server is not busy, that is, the rates of messages flowing into and out of a destination are low, then messages might be accumulating because of not being acknowledged.

Check for the message flow rate into and out of the broker:

```
imqcmd metrics bkr -m rts
```

Then check flow rates for each of the individual destinations:

```
imqcmd metrics bkr -t destType -n destName -m rts
```

Also check client code to see if messages are being properly acknowledged.

- There are inactive durable subscriptions on a topic destination.

  If a durable subscription is inactive, then messages are stored in a destination until the corresponding consumer becomes active and can consume the messages.

  To confirm this cause of the problem:

  Check the state of durable subscriptions on each topic destination:

  ```
  imqcmd list dur -d destName
  ```

  To resolve the problem:

  You can take any of the following actions:

  - Purge all messages for the offending durable subscriptions (see "Managing Durable Subscriptions" on page 179).
  - Specify message limit and limit behavior attributes for the topic (see Table 6-10 on page 171). For example, you can specify the `REMOVE_OLDEST` and `REMOVE_LOW_PRIORITY` limit behaviors, which delete messages that accumulate in memory.
  - Purge all messages from the corresponding destinations (see "Purging Destinations" on page 176).
  - Limit the time messages can remain in memory: you can rewrite the producing client to set a time-to-live value on each message. You can override any such settings for all producers sharing a connection by setting the `imqOverrideJMSExpiration` and `imqJMSExpiration` connection factory attributes (see Table 7-3 on page 187).

- There are too few consumers available to consume messages in a queue.

  If there are too few active consumers to which messages can be delivered, a queue destination can become backlogged as messages accumulate. This condition can occur for any of the following reasons:

  - Too few active consumers exist for the destination.
  - Consuming clients have failed to establish connections.
  - no active consumers use a selector that matches messages in the queue.

  To confirm this cause of the problem:

  To help determine the reason for unavailable consumers, check the number of active consumers on a destination:

  ```
  imqcmd metrics dst -n destName -t q -m con
  ```

  To resolve the problem:

  You can take any of the following actions, depending on the reason for unavailable consumers:

  - Create more active consumers for the queue, by starting up additional consuming clients.
  - Adjust the imq.consumerFlowLimit broker property to optimize queue delivery to multiple consumers (see "Multiple Consumer Queue Performance" on page 286).
  - Specify message limit and limit behavior attributes for the queue (see Table 6-10 on page 171). For example, you can specify the `REMOVE_OLDEST` and `REMIOVE_LOW_PRIOROTY` limit behaviors, which delete messages that accumulate in memory.
  - Purge all messages from the corresponding destinations (see "Purging Destinations" on page 176).
  - Limit the time messages can remain in memory: you can rewrite the producing client to set a time-to-live value on each message, you can override any such setting for all producers sharing a connection by setting the `imqOverrideJMSExpiration` and `imqJMSExpiration` connection factory attributes (see Table 7-3 on page 187).

- Message consumers are processing too slowly to keep up with message producers.

In this case topic subscribers or queue receivers are consuming messages more slowly than the producers are sending messages. One or more destinations is getting backlogged with messages due to this imbalance.

To confirm this cause of the problem:

Check for the rate of flow of messages into and out of the broker:

```
imqcmd metrics bkr -m rts
```

Then check flow rates for each of the individual destinations:

```
imqcmd metrics bkr -t destType -n destName -m rts
```

To resolve the problem:

- Optimize consuming client code.
- For queue destinations, increase the number of active consumers (see "Multiple Consumer Queue Performance" on page 286).

- Client acknowledgement processing is slowing down message consumption.

  Two factors affect the processing of client acknowledgements:

  - Significant broker resources can be consumed in processing client acknowledgements. As a result, message consumption might be slowed in those acknowledgement modes in which consuming clients block until the broker confirms client acknowledgements.
  - JMS payload messages and MQ control messages (such as client acknowledgements) share the same connection. As a result, control messages can be held up by JMS payload messages, slowing message consumption.

  To confirm this cause of the problem:

  Check the flow of messages relative to the flow of packets. If the number of packets per second is out of proportion to the number of messages, then client acknowledgements might be a problem.

  Also check if the client has received a `JMSException [C4000]: Packet acknowledge failed` message.

  To resolve the problem:

  - Modify the acknowledgement mode used by clients, for example, switch to `DUPS_OK_ACKNOWLEDGEMENT` or `CLIENT_ACKNOWLEDGEMENT`.
  - If using `CLIENT_ACKNOWLEDGEMENT` or transacted sessions, group a larger number of messages into a single acknowledgement.

- Adjust consumer and connection flow control parameters (see "Client Runtime Message Flow Adjustments" on page 286).

- The broker is not able to keep up with produced messages.

  In this case, messages are flowing into the broker faster than the broker can route and dispatch them to consumers. The sluggishness of the broker can be due to limitations in any or all of the following: cpu, network socket read/write operations, disk read/write operations, memory paging, the persistent store, or JVM memory limits.

  To confirm this cause of the problem:

  Check that none of the other causes of this problem are responsible.

  To resolve the problem:

  - Upgrade the speed of your computer or your data store.
  - Use a broker cluster to distribute the load among a number of broker instances.

Problem: Message Server Throughput Is Sporadic

Symptoms:

- Message throughput sporadically drops, then resumes normal performance.

Possible Causes:

- The broker is very low on memory resources

  Because destination and broker limits were not properly set, the broker takes increasingly serious action to prevent memory overload, and this can cause the broker to become very sluggish until the message backlog is cleared.

  To confirm this cause of the problem:

  Check the broker log for a low memory condition (`[B1089]: In low memory condition, broker is attempting to free up resources`), followed by an entry describing the new memory state and the amount of total memory being used.

  Also check the free memory available in the JVM heap:

```
imqcmd metrics bkr -m cxn
```

Free memory is low when the value of total JVM memory is close to the maximum JVM memory value.

To resolve the problem:

- Adjust the JVM (see "Java Virtual Machine Adjustments" on page 281).
- Increase system swap space.

- JVM memory reclamation (garbage collection) is taking place.

  Memory reclamation periodically sweeps through the system to free up memory. When this occurs, all threads are blocked. The larger the amount of memory to be freed up and the larger the JVM heap size, the larger the delay due to memory reclamation.

  To confirm this cause of the problem:

  Monitor cpu usage on your computer. There will be a big drop when memory reclamation is taking place.

  Also start your broker using the following command line options:

  ```
  -vmargs -verbose:gc
  ```

  Standard output indicates the time that memory reclamation takes place.

  To resolve the problem:

  In multiple cpu computers, set the memory reclamation to take place in parallel:

  ```
  -XX:+UseParallelGC=true
  ```

- The JVM is using the Just-In-Time compiler to speed up performance.

  To confirm this cause of the problem:

  Check that none of the other causes of this problem are responsible.

  To resolve the problem:

  Let the system run for a while; performance should improve.

Problem: Messages Not Reaching Consumers

Symptoms:

- Messages sent by producers are not received by consumers.

Possible Causes:

- Limit behaviors are causing messages to be deleted on the broker.

  When the number of messages or number of message bytes in destination memory reach configured limits, the broker will attempt to conserve memory resources. Three of the configurable behaviors taken by the broker when these limits are reached will cause messages to be lost:

  - `REMOVE_OLDEST`: deleting the oldest messages
  - `REMOVE_LOW_PRIORITY`: deleting the lowest priority messages according to age of the messages
  - `REJECT_NEWEST`: rejecting new messages (which throws an exception for rejected persistent messages).

  As the number of messages or number of message bytes in broker memory reach configured limits, the broker will attempt to conserve memory resources by rejecting the newest messages.

  To confirm this cause of the problem:

  Check the broker log for the following entry: `WARNING [B2011]: Storing of JMS message from…failed`. This entry will be followed by another entry explaining the limit that has been exceeded. There will be no entry, however, showing the deletion of messages.

  To resolve the problem:

  Change limits or change behavior.

- Message timeout value expires

  The broker will delete messages whose timeout value has expired. If a destination gets sufficiently backlogged with messages, messages whose time-to-live value is too short might be deleted.

  To confirm this cause of the problem:

  Check broker log file for the following entry; `Expiring Messages: Expired` *n* `messages.`

To resolve the problem:

Use override

- Clock times are not synchronized between different computers.

  If clocks are not in synchronization, then broker calculations of message lifetimes can be in error, causing messages to exceed their expiration time and be deleted.

  To confirm this cause of the problem:

  Check clocks on all computers.

  To resolve the problem:

  Synchronize clocks (see "System Clock Settings" on page 337).

- Consuming client failed to start message delivery on a connection.

  Messages cannot be delivered until client code establishes a connection and starts message delivery on the connection.

  To confirm this cause of the problem:

  Check that client code establishes a connection and starts message delivery.

  To resolve the problem:

  Rewrite the client code to establish a connection and start message delivery.

Adjusting Your Configuration To Improve Performance

System Adjustments

The following sections describe adjustments you can make to the operating system, JVM, and communication protocols.

Solaris Tuning: CPU Utilization, Paging/Swapping/Disk I/O

See your system documentation for tuning your operating system.

Java Virtual Machine Adjustments

By default, the broker uses a JVM heap size of 192MB. This is often too small for significant message loads and should be increased.

When the broker gets close to exhausting the JVM heap space used by Java objects, it uses various techniques such as flow control and message swapping to free memory. Under extreme circumstances it even closes client connections in order to free the memory and reduce the message inflow. Hence it is desirable to set the maximum JVM heap space high enough to avoid such circumstances.

However, if the maximum Java heap space is set too high, in relation to system physical memory, the broker can continue to grow the Java heap space until the entire system runs out of memory. This can result in diminished performance, unpredictable broker crashes, and/or affect the behavior of other applications and services running on the system. In general, you need to allow enough physical memory for the operating system and other applications to run on the machine.

In general it is a good idea to evaluate the normal and peak system memory footprints, and configure the Java heap size so that it is large enough to provide good performance, but not so large as to risk system memory problems.

To change the minimum and maximum heap size for the broker, use the `-vmargs` command line option when starting the broker. For example:

```
/usr/bin/imqbrokerd -vmargs "-Xms256m -Xmx1024m"
```

This command will set the starting Java heap size to 256MB and the maximum Java heap size to 1GB.

- On Solaris, if starting the broker via `/etc/rc` (that is, `/etc/init.d/imq`), specify broker command line arguments in the `/etc/imq/imqbrokerd.conf` file. See the comments in that file for more information.
- On Windows, if starting the broker as a Window's service, specify JVM arguments using the `-vmargs` option to the `imqsvcadmin install` command. See "Service Administrator Utility (imqsvcadmin)" on page 332.

In any case, verify settings by checking the broker's log file or using the `imqcmd metrics bkr -m cxn` command.

Tuning Transport Protocols

Once a protocol that meets application needs has been chosen, additional tuning (based on the selected protocol) might improve performance.

A protocol's performance can be modified using the following three broker properties:

- `imq.protocol` *protocol_type* `nodelay`
- `imq.protocol` *protocol_type* `inbufsz`
- `imq.protocol` *protocol_type* `outbufsz`

For TCP and SSL protocols, these properties affect the speed of message delivery between client and broker. For HTTP and HTTPS protocols, these properties affect the speed of message delivery between the MQ tunnel servlet (running on a Web server) and the broker. For HTTP/HTTPS protocols there are additional properties that can affect performance (see "HTTP/HTTPS Tuning" on page 284).

The protocol tuning properties are described in the following sections.

nodelay

The `nodelay` property affects Nagle's algorithm (the value of the TCP_NODELAY socket-level option on TCP/IP) for the given protocol. Nagle's algorithm is used to improve TCP performance on systems using slow connections (for example, wide-area networks (WANs).

When the algorithm is used, TCP tries to prevent several small chunks of data from being sent to the remote system (by bundling the data in larger packets). If the data written to the socket does not fill the required buffer size, the protocol delays sending the packet until either the buffer is filled or a specific delay time has elapsed. Once the buffer is full or the time-out has occurred, the packet is sent.

For most messaging applications, performance is best if there is no delay in the sending of packets (Nagle's algorithm is not enabled). This is because most interactions between client and broker are request/response interactions: the client sends a packet of data to the broker and waits for a response. For example, typical interactions include:

- Creating a connection
- Creating a producer or consumer
- Sending a persistent message (the broker confirms receipt of the message)
- Sending a client acknowledgement in an `AUTO_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE` session (the broker confirms processing of the acknowledgement)

For these interactions, most packets are smaller than the buffer size. This means that if Nagle's algorithm is used, the broker delays several milliseconds before sending a response to the consumer.

However, Nagle's algorithm may improve performance in situations where connections are slow and broker responses are not required. This would be the case where a client sends a non-persistent message or where a client acknowledgement is not confirmed by the broker (`DUPS_OK_ACKNOWLEDGE` session).

inbufsz/outbufsz

The `inbufsz` property sets the size of the buffer on the input stream reading data coming in from a socket. Similarly, `outbufsz` sets the buffer size of the output stream used by the broker to write data to the socket.

In general, both parameters should be set to values that are slightly larger than the average packet being received or sent. A good rule of thumb is to set these property values to the size of the average packet plus 1k (rounded to the nearest k).

For example, if the broker is receiving packets with a body size of 1k, the overall size of the packet (message body + header + properties) is about 1200 bytes. An `inbufsz` of 2k (2048 bytes) gives reasonable performance.

Increasing the inbufsz or outbufsz greater than that size may improve performance slightly; however, it increases the memory needed for each connection.

Figure 9-6 shows the consequence of changing `inbufsz` on a 1k packet.

Figure 9-7 Effect of Changing `inbufsz` on a 1k (1024 bytes) Packet

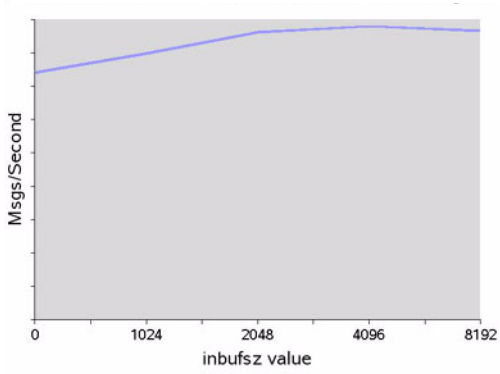
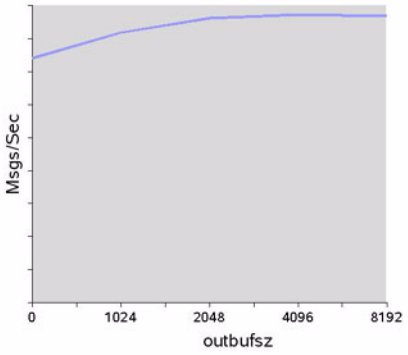


Figure 9-8 shows the consequence of changing `outbufsz` on a 1k packet.

Figure 9-8 Effect of Changing `outbufsz` on a 1k (1024 bytes) Packet



HTTP/HTTPS Tuning

In addition to the general properties discussed in the previous two sections, HTTP/HTTPS performance is limited by how fast a client can make HTTP requests to the Web server hosting the MQ tunnel servlet.

A Web server might need to be optimized to handle multiple requests on a single socket. With JDK version 1.4 and later, HTTP connections to a Web server are kept alive (the socket to the Web server remains open) to minimize resources used by the Web server when it processes multiple HTTP requests. If the performance of a client application using JDK version 1.4 is slower than the same application running with an earlier JDK release, you might need to tune the Web server keep-alive configuration parameters to improve performance.

In addition to such Web-server tuning, you can also adjust how often a client polls the Web server. HTTP is a request-based protocol. This means that clients using an HTTP-based protocol periodically need to check the Web server to see if messages are waiting. The `imq.httpjms.http.pullPeriod` broker property (and the corresponding `imq.httpsjms.https.pullPeriod` property) specifies how often the MQ client runtime polls the Web server.

If the `pullPeriod` value is `-1` (the default value), the client runtime polls the server as soon as the previous request returns, maximizing the performance of the individual client. As a result, each client connection monopolizes a request thread in the Web server, possibly straining Web server resources.

If the `pullPeriod` value is a positive number, the client runtime periodically sends requests to the Web server to see if there is pending data. In this case, the client does not monopolize a request thread in the Web server. Hence, if large numbers of clients are using the Web server, you might conserve Web server resources by setting the `pullPeriod` to a positive value.

Tuning the File-based Persistent Store

For information on tuning the file-based persistent store, see "Built-in persistence" on page 64.

Broker Adjustments

The following sections describe adjustments you can make to broker properties to improve performance.

Memory Management: Increasing Broker Stability Under Load

Memory management can be configured on a destination-by-destination level or on a system-wide level (for all destinations, collectively).

Using Destination Limits

For information on destination limits, see "Managing Destinations" on page 168.

Using System-wide Limits

If message producers tend to overrun message consumers, then messages can accumulate in the broker. While the broker does contain a mechanism for throttling back producers and swapping messages out of active memory in low memory conditions (see "Managing Memory Resources and Message Flow" on page 61), it's wise to set a hard limit on the total number of messages (and message bytes) that the broker can hold.

Control these limits by setting the `imq.system.max_count` and the `imq.system.max_size` broker properties. See "Editing the Instance Configuration File" on page 129 or "Summary of imqbrokerd Options" on page 136 for information on setting broker properties.

For example

```
imq.system.max_count=5000
```

The defined value above means that the broker will only hold up to 5000 undelivered/unacknowledged messages. If additional messages are sent, they are rejected by the broker. If a message is persistent then the producer will get an exception when it tries to send the message. If the message is non-persistent, then the broker silently drops the message.

To have non-persistent messages return an exception like persistent messages, set the following property on the connection factory object used by the client:

```
imqAckOnProduce = true
```

The setting above may decrease the performance of sending non-persistent messages to the broker (the client waits for a reply before sending the next message), but often this is acceptable since message inflow to the broker is typically not a system bottleneck.

When an exception is returned in sending a message, the client should pause for a moment and retry the send again.

Multiple Consumer Queue Performance

The efficiency with which multiple queue consumers process the messages in a queue destination depends on configurable queue destination attributes, namely the number of active consumers (`maxNumActiveConsumers`) and the maximum number of messages that can be delivered to a consumer in a single batch (`consumerFlowLimit`). These attributes are described in Table 6-10 on page 171.

To achieve optimal message throughput there must be a sufficient number of active consumers to keep up with the rate of message production for the queue, and the messages in the queue must be routed and then delivered to the active consumers in such a way as to maximize their rate of consumption. The general mechanism for balancing message delivery among multiple consumers is described in "Queue Delivery to Multiple Consumers" on page 76.

If messages are accumulating in the queue, it is possible that there is an insufficient number of active consumers to handle the message load. It is also possible that messages are being delivered to the consumers in batch sizes that cause messages to be backing up on the consumers. For example, if the batch size (`consumerFlowLimit`) is too large, one consumer might receive all the messages in a queue while other active consumers receive none. If consumers are very fast, this might not be a problem.

However, if consumers are relatively slow, you want messages to be distributed to them evenly, and therefore you want the batch size to be small. The smaller the batch size, the more overhead is required to deliver messages to consumers. Nevertheless, for slow consumers, there is generally a net performance gain to using small batch sizes.

Client Runtime Message Flow Adjustments

This section discusses flow control behaviors that impact performance (see "Client Runtime Configuration" on page 243). These behaviors are configured as attributes of connection factory administered objects. For information on setting connection factory attributes, see Chapter 7, "Managing Administered Objects."

Message Flow Metering

Messages sent and received by clients (JMS messages), as well as MQ control messages, pass over the same client-broker connection. Delays in the delivery of control messages, such as broker acknowledgements, can result if control messages are held up by the delivery of JMS messages. To prevent this type of congestion, MQ meters the flow of JMS messages across a connection.

JMS messages are batched (as specified with the `imqConnectionFlowCount` property) so that only a set number are delivered; when the batch has been delivered, delivery of JMS messages is suspended, and pending control messages are delivered. This cycle repeats, as other batches of JMS messages are delivered, followed by queued up control messages.

The value of `imqConnectionFlowCount` should be kept low if the client is doing operations that require many responses from the broker; for example, the client is using the `CLIENT_ACKNOWLEDGE or AUTO_ACKNOWLEDGE` modes, persistent messages, transactions, queue browsers, or if the client is adding or removing consumers. If, on the other hand, the client has only simple consumers on a connection using `DUPS_OK_ACKNOWLEDGE` mode, you can increase `imqConnectionFlowCount` without compromising performance.

Message Flow Limits

There is a limit to the number of JMS messages that the MQ client runtime can handle before encountering local resource limitations, such as memory. When this limit is approached, performance suffers. Hence, MQ lets you limit the number of messages per consumer (or messages per connection) that can be delivered over a connection and buffered in the client runtime, waiting to be consumed.

Consumer-based Limits

When the number of JMS messages delivered to the client runtime exceeds the value of `imqConsumerFlowLimit` for any consumer, message delivery for that consumer stops. It is resumed only when the number of unconsumed messages for that consumer drops below the value set with `imqConsumerFlowThreshold`.

The following example illustrates the use of these limits: consider the default settings for topic consumers

```
imqConsumerFlowLimit=1000
```

```
imqConsumerFlowThreshold=50
```

When the consumer is created, the broker delivers an initial batch of 1000 messages (providing they exist) to this consumer without pausing. After sending 1000 messages, the broker stops delivery until the client runtime asks for more messages. The client runtime holds these messages until the application processes them. The client runtime then allows the application to consume at least 50% (`imqConsumerFlowThreshold`) of the message buffer capacity (i.e. 500 messages) before asking the broker to send the next batch.

In the same situation, if the threshold were 10%, the client runtime would wait for the application to consume at least 900 messages before asking for the next batch.

The next batch size is calculated as follows:

`imqConsumerFlowLimit -` (*current number of pending msgs in buffer*)

So, if `imqConsumerFlowThreshold` is 50%, the next batch size can fluctuate between 500 and 1000, depending on how fast the application can process the messages.

If the `imqConsumerFlowThreshold` is set too high (close to 100%), the broker will tend to send smaller batches, which can lower message throughput. If the value is set too low (close to 0%), the client might be able to finish processing the remaining buffered messages before the broker delivers the next set, causing message throughput degradation. Generally speaking, unless you have specific performance or reliability concerns, you will not have to change the default value of `imqConsumerFlowThreshold` attribute.

The consumer-based flow controls (in particular `imqConsumerFlowLimit`) are the best way to manage memory in the client runtime. Generally, depending on the client application, you know the number of consumers you need to support on any connection, the size of the messages, and the total amount of memory that is available to the client runtime.

Connection-based Limits

In the case of some client applications, however, the number of consumers might be indeterminate, depending on choices made by end users. In those cases, you can still manage memory, using connection-level flow limits.

Connection-level flow controls limit the total number of messages buffered for *all* consumers on a connection. If this number exceeds the `imqConnectionFlowLimit`, then delivery of messages through the connection will stop until that total drops below the connection limit. (The `imqConnectionFlowLimit` is only enabled if you set the `imqConnectionFlowLimitEnabled` property to `true`.)

The number of messages queued up in a session is a function of the number of message consumers using the session and the message load for each consumer. If a client is exhibiting delays in producing or consuming messages, you can normally improve performance by redesigning the application to distribute message producers and consumers among a larger number of sessions or to distribute sessions among a larger number of connections.

Appendix A

Location of MQ Data

Sun™ ONE Message Queue (MQ) uses many categories of data, each of which is stored in a different location, depending on the operating system, as shown in the following sections. In the tables that follow, *instanceName* identifies the name of the broker instance with which the data is associated.

Solaris

Table A-1 shows the location of MQ data on the Solaris platform.

NOTE Data locations for MQ bundled with Sun ONE Application Server, on Solaris are shown in Table A-3 on page 293.

Table A-1 Location of MQ Data on Solaris

Data Category	Location on Solaris
Broker instance configuration properties	`/var/imq/instances/`*instanceName*`/props/config.properties`
Broker configuration file templates	`/usr/share/lib/imq/props/broker/`
Persistent store (messages, destinations, durable subscriptions, transactions)	`/var/imq/instances/`*instanceName*`/fs350/` or a JDBC-accessible data store
Broker instance log file directory (default location)	`/var/imq/instances/`*instanceName*`/log/`
Administered objects (object store)	local directory of your choice or an LDAP server

Table A-1 Location of MQ Data on Solaris *(Continued)*

Data Category	Location on Solaris
Security: user repository	`/var/imq/instances/`*instanceName*`/etc/passwd` or an LDAP server
Security: access control file (default location)	`/var/imq/instances/`*instanceName*`/etc/accesscontrol.properties`
Security: passfile directory (default location)	`/var/imq/instances/`*instanceName*`/etc/`
Security: example passfile	`/etc/imq/passfile.sample`
Security: broker's keystore file location	`/etc/imq/`
JavaDoc API documentation	`/usr/share/javadoc/imq/index.html`
Example applications and configurations	`/usr/demo/imq/`
Java archive (`.jar`), web archive (`.war`), and resource adapter archive (`.rar`) files	`/usr/share/lib/`

Linux

Table A-2 shows the location of MQ data on the Linux platform.

Table A-2 Location of MQ Data on Linux

Data Category	Location on Windows
Broker instance configuration properties	`/var/opt/imq/instances/`*instanceName*`/props/config.properties`
Broker configuration file templates	`/opt/imq/lib/props/broker/`
Persistent store (messages, destinations, durable subscriptions, transactions)	`/var/opt/imq/instances/`*instanceName*`/fs350/` or a JDBC-accessible data store
Broker instance log file directory (default location)	`/var/opt/imq/instances/`*instanceName*`/log/`
Administered objects (object store)	local directory of your choice or an LDAP server

Table A-2 Location of MQ Data on Linux *(Continued)*

Data Category	Location on Windows
Security: user repository	`/var/opt/imq/instances/`*instanceName*`/etc/passwd`
	or an LDAP server
Security: access control file (default location)	`/var/opt/imq/instances/`*instanceName*`/etc/accesscontrol.properties`
Security: passfile directory (default location)	`/var/opt/imq/instances/`*instanceName*`/etc/`
Security: example passfile	`/etc/opt/imq/passfile.sample`
Security: broker's keystore file location	`/etc/opt/imq/`
JavaDoc API documentation	`/opt/imq/javadoc/index.html`
Example applications and configurations	`/opt/imq/demo/`
Java archive (`.jar`), web archive (`.war`), and resource adapter archive (`.rar`) files	`/opt/imq/lib/`

Windows

Table A-3 shows the location of MQ data on the Windows platform and on some MQ installations bundled with Sun ONE Application Server. For more information, see Table 3 on page 26, and the definitions for `IMQ_HOME` and `IMQ_VARHOME`.

Table A-3 Location of MQ Data on Windows

Data Category	Location on Windows
Broker instance configuration properties	`IMQ_VARHOME\instances\`*instanceName*`\props\config.properties`
Broker configuration file templates	`IMQ_HOME\lib\props\broker\`
Persistent store (messages, destinations, durable subscriptions, transactions)	`IMQ_VARHOME\instances\`*instanceName*`\fs350\`
	or a JDBC-accessible data store
Broker instance log file directory (default location)	`IMQ_VARHOME\instances\`*instanceName*`\log\`

Table A-3 Location of MQ Data on Windows *(Continued)*

Data Category	Location on Windows
Administered objects (object store)	local directory of your choice or an LDAP server
Security: user repository	`IMQ_VARHOME\instances\`*instanceName*`\etc\passwd` or an LDAP server
Security: access control file (default)	`IMQ_VARHOME\instances\`*instanceName*`\etc\accesscontrol.properties`
Security: passfile directory (default location)	`IMQ_HOME\etc\`
Security: example passfile	`IMQ_HOME\etc\passfile.sample`
Security: broker's keystore file location	`IMQ_HOME\etc\`
JavaDoc API documentation	`IMQ_HOME\javadoc\index.html`
Example applications and configurations	`IMQ_HOME\demo\`
Java archive (`.jar`), web archive (`.war`), and resource adapter archive (`.rar`) files	`IMQ_HOME\lib\`

Appendix B

Setting Up Plugged-in Persistence

This appendix explains how to set up a broker to use plugged-in persistence to access a JDBC-accessible data store.

Introduction

Sun™ ONE Message Queue (MQ) brokers include a Persistence Manager component that manages the writing and retrieval of persistent information (see "Persistence Manager" on page 63). The Persistence Manager is configured by default to access a built-in, file-based data store, but you can reconfigure it to plug in any data store accessible through a JDBC-compliant driver.

To configure a broker to use plugged-in persistence, you need to set a number of JDBC-related properties in the broker instance configuration file. You also need to create the appropriate database schema for performing MQ persistence operations. MQ provides a utility, Database Manager (`imqdbmgr`), which uses your JDBC driver and broker configuration properties to create and manage the plugged-in database.

The procedure described in this appendix is illustrated using, as an example, the PointBase DBMS bundled with the Java 2 Platform, Enterprise Edition (J2EE) SDK. Version 1.4 is available for download from `java.sun.com`. The example uses PointBase's embedded version (instead of the client/server version). In the procedures, instructions are illustrated using path names and property names from the PointBase example. They are identified by the word "Example:"

Example configurations for Oracle and PointBase can be found in the examples location shown in Appendix A, "Location of MQ Data." In addition, examples for PointBase embedded version, PointBase server version, Oracle, and Cloudscape are provided as commented-out values in the instance configuration file.

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 B-1 on page 298.

2. Place a copy or a symbolic link to your JDBC driver jar file located in the following path:

   `/usr/share/lib/imq/ext/` (on Solaris)

   `/opt/imq/lib/ext/` (on Linux)

   `IMQ_VARHOME\lib\ext` (on Windows)

   *Copy Example (*Solaris*):*

   ```
   % cp j2eeSDK_install_directory/pointbase/lib/pointbase.jar
   /usr/share/lib/imq/ext
   ```

   Symbolic Link Example (Solaris):

   ```
   % ln -s j2eeSDK_install_directory/lib/pointbase/pointbase.jar
   /usr/share/lib/imq/ext
   ```

3. Create the database schema needed for MQ persistence.

   Use the `imqdbmgr create all` command (for an embedded database) or the `imqdbmgr create tbl` command (for an external database). See "Database Manager Utility (imqdbmgr)" on page 301.

   Example:

   a. Change to directory where `imqdbmgr` resides.

      `cd /usr/bin` (on Solaris)

      `cd /opt/imq/bin` (on Linux)

      `cd IMQ_HOME/bin` (on Windows)

   b. enter the `imqdbmgr` command.

      ```
      imqdbmgr create all
      ```

NOTE If an embedded database is used, it is recommended that it be created under the following directory:

…/instances/*instanceName*/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 user who runs the broker should be the same user who created the embedded database using the `imqdbmgr` command (see "Database Manager Utility (imqdbmgr)" on page 301).

JDBC-related Broker Configuration Properties

The broker's instance configuration file is located in a directory identified by the name of the broker instance (*instanceName*) with which the configuration file is associated (see Appendix A, "Location of MQ Data"):

…/instances/*instanceName*/props/config.properties

If the file does not yet exist, you have to start up the broker using the `-name` *instanceName* option, for MQ to create the file.

Table B-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 that uses plugged-in persistence.

The instance configuration properties enable you to customize the SQL code that creates the MQ database schema: there is a configurable property that specifies the SQL code that creates each database table. These properties are needed to properly specify the data types used by the plugged-in database.

Since there are incompatibilities between database vendors with respect to the exact SQL syntax, be sure to check the corresponding documentation from your database vendor and adjust the properties in Table B-1accordingly. For example, for the PointBase database, you may need to adjust the maximum length allowed for the MSG column (see the `imq.persist.jdbc.table.IMQMSG35` property) in the IMQMSG35 table.

Table B-1 includes values you would specify for the PointBase DBMS example.

Table B-1 JDBC-related Properties

Property Name	Description
`imq.persist.store`	Specifies a file-based or JDBC-based data store. *Example*: `jdbc`
`imq.persist.jdbc.brokerid` `(optional)`	Specifies a broker instance identifier that is appended to database table names to make them unique in the case where more than one broker instance is using the same database as a persistent data store. (Usually not needed in the case of an embedded database, which stores data for only one broker instance.) The identifier must be an alphanumeric string whose length does not exceed the maximum table name length, minus 12, allowed by the database. *Example:* not needed for PointBase embedded version.
`imq.persist.jdbc.driver`	Specifies the java class name of the JDBC driver to connect to the database. *Example*: `com.pointbase.jdbc.jdbcUniversalDriver`

`imq.persist.jdbc.opendburl`	Specifies the database URL for opening a connection to an existing database.
	Example:
	`jdbc:pointbase:embedded:`*dbName*`;`
	`database.home=` …`/instances/`*instanceName*`/dbstore`

`imq.persist.jdbc.createdburl` `(optional)`	Specifies the database URL for opening a connection to create a database. (Only specified if the database is to be created using `imqdbmgr`.)
	Example:
	`jdbc:pointbase:embedded:`*dbName*`;new,`
	`database.home=` …`/instances/`*instanceName*`/dbstore`

`imq.persist.jdbc.closedburl` `(optional)`	Specifies the database URL for shutting down the current database connection when the broker is shutdown. *Example:* not required for PointBase
`imq.persist.jdbc.user` `(optional)`	Specifies the user name used to open a database connection, if required. For security reasons, the value can be specified instead using command line options: `imqbrokerd -dbuser` and `imqdbmgr -u`

Table B-1 JDBC-related Properties *(Continued)*

Property Name	Description

`imq.persist.jdbc.needpassword (optional)`	Specifies whether the database requires a password for broker access. Value of `true` means password is required. The password can be specified using the following command line options:
	`imqbrokerd -dbpassword`
	`imqdbmgr -p`
	If the password is not provided using either command line options or a passfile (see "Using a Passfile" on page 224), the broker will prompt for the password.

`imq.persist.jdbc.password (optional)`	Specifies password used to open a database connection, if required. This property can only be specified in a passfile (see "Using a Passfile" on page 224).
	There are a number of ways to provide a password. The most secure is to let the broker prompt you for a password. Less secure is to use a passfile and read-protect the passfile. Least secure is to specify the password using the following command line options:
	`imqbrokerd -dbpassword`
	`imqdbmgr -p`

`imq.persist.jdbc.table.IMQSV35`	SQL command used to create the version table. *Example*: `CREATE TABLE ${name} (STOREVERSION INTEGER NOT NULL, BROKERID VARCHAR(100))`
`imq.persist.jdbc.table.IMQCCREC35`	SQL command used to create the configuration change record table. *Example*: `CREATE TABLE ${name} (RECORDTIME BIGINT NOT NULL, RECORD BLOB(10k))`
`imq.persist.jdbc.table.IMQDEST35`	SQL command used to create the destination table. *Example*: `CREATE TABLE ${name} (DID VARCHAR(100) NOT NULL, DEST BLOB(10k), primary key(DID))`
`imq.persist.jdbc.table.IMQINT35`	SQL command used to create the interest table. *Example*: `CREATE TABLE ${name} (CUID BIGINT NOT NULL, INTEREST BLOB(10k), primary key(CUID))`

Table B-1 JDBC-related Properties *(Continued)*

Property Name	Description
imq.persist.jdbc.table.IMQMSG35	SQL command used to create the message table. *Example*: `CREATE TABLE ${name} (MID VARCHAR(100) NOT NULL, DID VARCHAR(100), MSGSIZE BIGINT, MSG BLOB(1m), primary key(MID))` The default maximum length for the MSG column is `1 Megabyte` (`1m`). If you expect to have messages that are larger than this, set the length accordingly. If the tables have already been created, you need to recreate them to make the change.
imq.persist.jdbc.table.IMQPROPS35	SQL command used to create the property table. *Example*: `CREATE TABLE ${name} (PROPNAME VARCHAR(100) NOT NULL, PROPVALUE BLOB(10k), primary key(PROPNAME))`
imq.persist.jdbc.table.IMQILIST35	SQL command used to create the interest state table. *Example*: `CREATE TABLE ${name} (MID VARCHAR(100) NOT NULL, CUID BIGINT, DID VARCHAR(100), STATE INTEGER, primary key(MID, CUID))`
imq.persist.jdbc.table.IMQTXN35	SQL command used to create the transaction table. *Example*: `CREATE TABLE ${name} (TUID BIGINT NOT NULL, STATE INTEGER, TSTATEOBJ BLOB(10K), primary key(TUID))`
imq.persist.jdbc.table.IMQTACK35	SQL command used to create the transaction acknowledgement table. *Example*: `CREATE TABLE ${name} (TUID BIGINT NOT NULL, TXNACK BLOB(10k))`

As with all broker configuration properties, values can be set using the `-D` command line option. If a database requires certain database specific properties to be set, these also can be set using the `-D` command line option when starting the broker (`imqbrokerd`) or the Database Manager utility (`imqdbmgr`).

Example:

For the PointBase embedded database example, instead of specifying the absolute path of a database in database connection URLs (as those shown in Table B-1 examples), the -D command line option can be used to define the PointBase system directory:

```
-Ddatabase.home=IMQ_VARHOME/instances/instanceName/dbstore
```

In that case the URLs to create and open a database can be specified simply as:

```
imq.persist.jdbc.createdburl=jdbc:pointbase:embedded:dbName;new
```

and

```
imq.persist.jdbc.opendburl=jdbc:pointbase:embedded:dbName
```

respectively.

Database Manager Utility (imqdbmgr)

MQ provides a Database Manager utility (`imqdbmgr`) for setting up the schema needed for persistence. The utility can also be used to delete MQ database tables should the tables become corrupted or should you wish to use a different database as a data store.

This section describes the basic `imqdbmgr` command syntax, provides a listing of subcommands, and summarizes `imqdbmgr` command options.

Syntax of the imqdbmgr Command

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

```
imqdbmgr subcommand argument [options]
imqdbmgr -h|-help
imqdbmgr -v|-version
```

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

```
imqdbmgr create all -v
```

imqdbmgr Subcommands

The Database Manager utility (`imqdbmgr`) includes the subcommands listed in Table B-2:

Table B-2 `imqdbmgr` Subcommands

Subcommand and Argument	Description
`create all`	Creates a new database and MQ persistent store schema. This command is used on an embedded database system, and when used, the property `imq.persist.jdbc.createdburl` needs to be specified.
`create tbl`	Creates the MQ persistent store schema in an existing database system. This command is used on an external database system.
`delete tbl`	Deletes the existing MQ database tables in the current persistent store database.
`delete oldtbl`	Deletes all MQ database tables in an earlier version persistent store database. Used after the persistent store has been automatically migrated to the current version of MQ.
`recreate tbl`	Deletes the existing MQ database tables in the current persistent store database and then re-creates the MQ persistent store schema.
`reset lck`	Resets the lock so the persistent store database can be used by other processes.

Summary of imqdbmgr Command Options

Table B-3 lists the options to the `imqdbmgr` command.

Table B-3 `imqdbmgr` Options

Option	Description
`-D`*property=value*	Sets the specified property to the specified value.
`-b` *instanceName*	Specifies the broker instance name and use the corresponding instance configuration file.
`-h`	Displays usage help. Nothing else on the command line is executed.
`-p` *password*	Specifies the database password.
`-u` *name*	Specifies the database user name.
`-v`	Displays version information. Nothing else on the command line is executed.

Appendix C

HTTP/HTTPS Support (Enterprise Edition)

Message Queue, Enterprise Edition (see "Product Editions" on page 32) includes support for both HTTP and HTTPS connections. (HTTPS is HTTP over a Secure Socket Layer—SSL—transport connection.) This support allows client applications to communicate with the broker using the HTTP protocol instead of direct TCP connections. This appendix describes the architecture used to enable this support and explains the setup work needed to allow clients to use HTTP-based connections for MQ messaging.

NOTE HTTP/HTTPS support is available for Java clients but not for C clients.

HTTP/HTTPS Support Architecture

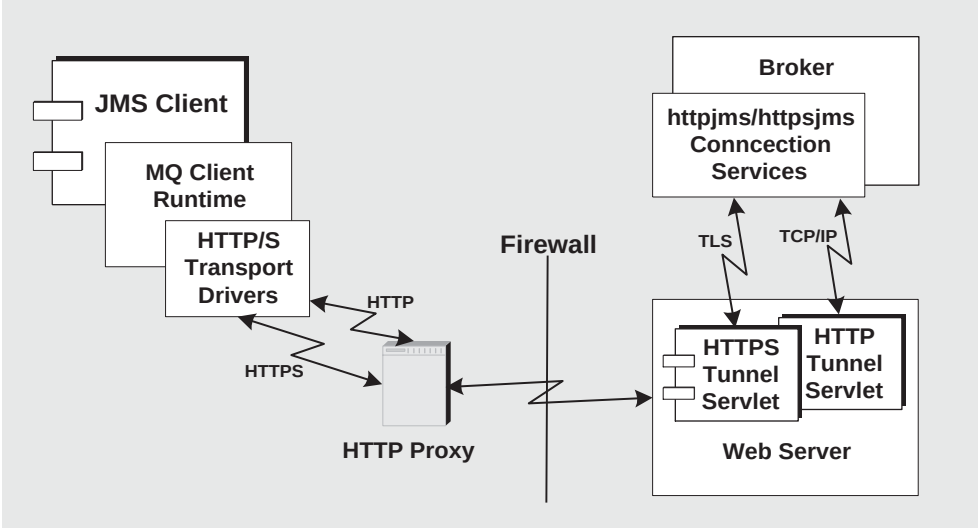
MQ messaging can be run on top of HTTP/HTTPS connections. Because HTTP/HTTPS connections are normally allowed through firewalls, this allows client applications to be separated from a broker by a firewall.

Figure C-1 on page 306 shows the main components involved in providing HTTP/HTTPS support.

- On the client side, an HTTP or HTTPS transport driver encapsulates the MQ message into an HTTP request and makes sure that these requests are sent to the Web server in the correct sequence.
- The client can use an HTTP proxy server to communicate with the broker if necessary. The proxy's address is specified using command line options when starting the client. See "Using an HTTP Proxy" on page 311 for more information.

- An HTTP or HTTPS tunnel servlet (both bundled with MQ) is loaded in the web server and used to pull JMS messages out of client HTTP requests before forwarding them to the broker. The HTTP/HTTPS tunnel servlet also sends broker messages back to the client in response to HTTP requests made by the client. A single HTTP/HTTPS tunnel servlet can be used to access multiple brokers.

Figure C-1 HTTP/HTTPS Support Architecture



- On the broker side, the httpjms or httpsjms connection service unwraps and de-multiplexes incoming messages from the corresponding tunnel servlet.
- If the Web server fails and is restarted, all connections are restored and there is no effect on clients. If the broker fails and is restarted, an exception is thrown and clients must re-establish their connections. In the unlikely case that both the Web server and the broker fail, and the broker is not restarted, the Web server will restore client connections and continue waiting for a broker connection— without notifying clients. To avoid this situation, always restart the broker.

As you can see from Figure C-1, the architecture for HTTP and HTTPS support are very similar. The main difference is that, in the case of HTTPS (httpsjms connection service), the tunnel servlet has a secure connection to both the client application and broker.

The secure connection to the broker is provided through an SSL-enabled tunnel servlet—MQ's HTTPS tunnel servlet—which passes a self-signed certificate to any broker requesting a connection. The certificate is used by the broker to set up an encrypted connection to the HTTPS tunnel servlet. Once this connection is established, a secure connection between a client application and the tunnel servlet can be negotiated by the client application and the web server.

Enabling HTTP Support

The following sections describe the steps you need to take to enable HTTP support.

➤ **To Enable HTTP Support**

1. Deploy the HTTP tunnel servlet on a web server.
2. Configure the broker's httpjms connection service and start the broker.
3. Configure an HTTP connection.

Step 1. Deploying the HTTP Tunnel Servlet on a Web Server

There are two general ways you can deploy the HTTP tunnel servlet on a web server:

- deploying it as a jar file—for web servers that support Servlet 2.1 or earlier
- deploying it as a web archive (WAR) file—for web servers that support Servlet 2.2 or later

Deploying as a Jar File

Deploying the MQ tunnel servlet consists of making the appropriate jar files accessible to the host web server, configuring the web server to load the servlet on startup, and specifying the context root portion of the servlet's URL.

The tunnel servlet jar file (`imqservlet.jar`) contains all the classes needed by the HTTP tunnel servlet, and can be found in a directory that depends upon operating system (see Appendix A, "Location of MQ Data").

Any web server with servlet 2.x support can be used to load this servlet. The servlet class name is:

```
com.sun.messaging.jmq.transport.
httptunnel.servlet.HttpTunnelServlet
```

The web server must be able to see the `imqservlet.jar` file. If you are planning to run the web server and the broker on different hosts, you should place a copy of the `imqservlet.jar` file in a location where the web server can access it.

You also need to configure the web server to load this servlet on startup, and you might need to specify the context root portion of the servlet's URL (see "Example 1: Deploying the HTTP Tunnel Servlet on Sun ONE Web Server" on page 312).

It is also recommended that you disable your web server's access logging feature in order to improve performance.

Deploying as a Web Archive File

Deploying the HTTP tunnel servlet as a WAR file consists of using the deployment mechanism provided by the web server. The HTTP tunnel servlet WAR file (`imqhttp.war`) is located in the directory containing .jar, .war, and .rar files, and depends on your operating system (see Appendix A, "Location of MQ Data").

The WAR file includes a deployment descriptor that contains the basic configuration information needed by the web server to load and run the servlet. Depending on the web server, you might also need to specify the context root portion of the servlet's URL (see "Example 2: Deploying the HTTP Tunnel Servlet on Sun ONE Application Server 7.0" on page 315).

Step 2. Configuring the httpjms Connection Service

HTTP support is not activated for a broker by default, so you need to reconfigure the broker to activate the httpjms connection service. Once reconfigured, the broker can be started as outlined in "Starting a Broker" on page 134.

➤ To Activate the httpjms Connection Service

1. Open the broker's instance configuration file.

   The instance configuration file is stored in a directory identified by the name of the broker instance (*instanceName*) with which the configuration file is associated (see Appendix A, "Location of MQ Data"):

   …/instances/*instanceName*/props/config.properties

2. Add the `httpjms` value to the `imq.service.activelist` property:

   ```
   imq.service.activelist=jms,admin,httpjms
   ```

At startup, the broker looks for a web server and HTTP tunnel servlet running on its host machine. To access a remote tunnel servlet, however, you can reconfigure the servletHost and servletPort connection service properties.

You can also reconfigure the pullPeriod property to improve performance. The httpjms connection service configuration properties are detailed in Table C-1 on page 309.

Table C-1 httpjms Connection Service Properties

Property Name	Description
`imq.httpjms.http.servletHost`	Change this value, if necessary, to specify the name of the host (hostname or IP address) on which the HTTP tunnel servlet is running. (This can be a remote host or a specific hostname on a local host.) Default: `localhost`
`imq.httpjms.http.servletPort`	Change this value to specify the port number that the broker uses to access the HTTP tunnel servlet. (If the default port is changed on the Web server, then you must change this property accordingly.) Default: 7675
`imq.httpjms.http.pullPeriod`	Specifies the interval, in seconds, between HTTP requests made by a client runtime to pull messages from the broker. (Note that this property is set on the broker and propagates to the client runtime.) If the value is zero or negative, the client keeps one HTTP request pending at all times, ready to pull messages as fast as possible. With a large number of clients, this can be a heavy drain on web server resources and the server may become unresponsive. In such cases, you should set the `pullPeriod` property to a positive number of seconds. This sets the time the client's HTTP transport driver waits before making subsequent pull requests. Setting the value to a positive number conserves web server resources at the expense of the response times observed by clients. Default: `-1`

Table C-1 httpjms Connection Service Properties *(Continued)*

Property Name	Description
`imq.httpjms.http.connectionTimeout`	Specifies the time, in seconds, that the client runtime waits for a response from the HTTP tunnel servlet before throwing an exception. (Note that this property is set on the broker and propagates to the client runtime.) This property also specifies the time the broker waits after communicating with the HTTP tunnel servlet before freeing up a connection. A timeout is necessary in this case because the broker and the tunnel servlet have no way of knowing if a client that is accessing the HTTP servlet has terminated abnormally. Default: `60`

Step 3. Configuring an HTTP Connection

A client application must use an appropriately configured connection factory administered object to make an HTTP connection to a broker. This section discusses HTTP connection configuration issues.

Configuring the Connection Factory

To enable HTTP support, you need to set the connection factory's `imqAddressList` attribute to the HTTP tunnel servlet URL. The general syntax of the HTTP tunnel servlet URL is the following:

`http://`*hostName:port/contextRoot*`/tunnel`

where *hostName:port* is the name and port of the web server hosting the HTTP tunnel servlet and *contextRoot* is a path set when deploying the tunnel servlet on the web server.

For more information on connection factory attributes in general, and the `imqAddressList` attribute in particular, see the MQ *Java Client Developer's Guide*.

You can set connection factory attributes in one of the following ways:

- Using the `-o` option to the `imqobjmgr` command that creates the connection factory administered object (see "Adding a Connection Factory" on page 195), or set the attribute when creating the connection factory administered object using the Administration Console (`imqadmin`).
- Using the `-D` option to the command that launches the client (see the MQ *Java Client Developer's Guide*).
- Using a an API call to set the attributes of a connection factory after you create it programmatically in client code (see the MQ *Java Client Developer's Guide*).

Using a Single Servlet to Access Multiple Brokers

You do not need to configure multiple web servers and servlet instances if you are running multiple brokers. You can share a single web server and HTTP tunnel servlet instance among concurrently running brokers. If multiple broker instances are sharing a single tunnel servlet, you must configure the `imqAddressList` connection factory attribute as shown below:

`http://`*hostName:port/contextRoot*`/tunnel?ServerName=`*bkrHostName*`:`*instanceName*

Where *bkrHostName* is the broker instance host name and *instanceName* is the name of the specific broker instance you want your client to access.

To check that you have entered the correct strings for *bkrHostName* and *instanceName,* generate a status report for the HTTP tunnel servlet by accessing the servlet URL from a browser. The report lists all brokers being accessed by the servlet:

```
HTTP tunnel servlet ready.
Servlet Start Time : Thu May 30 01:08:18 PDT 2002
Accepting TCP connections from brokers on port : 7675
Total available brokers = 2
Broker List :
   jpgserv:broker2
   cochin:broker1
```

Using an HTTP Proxy

If you are using an HTTP proxy to access the HTTP tunnel servlet:

- Set `http.proxyHost` system property to the proxy server host name.
- Set `http.proxyPort` system property to the proxy server port number.

You can set these properties using the `-D` option to the command that launches the client application.

Example 1: Deploying the HTTP Tunnel Servlet on Sun ONE Web Server

This section describes how you deploy the HTTP tunnel servlet both as a jar file and as a WAR file on the Sun ONE Web Server. The approach you use depends on the version of Sun ONE Web Server: If it does not support Servlet 2.2 or later, it will not be able to handle WAR file deployment.

Deploying as a Jar File

The instructions below refer to deployment on Sun ONE Web Server 6.1 using the browser-based administration GUI. This procedure consists of the following general steps:

1. add a servlet
2. configure the servlet virtual path
3. load the servlet
4. disable the servlet access log

These steps are described in the following subsections. You can verify successful HTTP tunnel servlet deployment by accessing the servlet URL using a web browser. It should display status information.

Adding a Servlet

➤ **To Add a Tunnel Servlet**

1. Select the Servlets tab.
2. Choose Configure Servlet Attributes.
3. Specify a name for the tunnel servlet in the Servlet Name field.
4. Set the Servlet Code (class name) field to the following value:

   ```
   com.sun.messaging.jmq.transport.
   httptunnel.servlet.HttpTunnelServlet
   ```

5. Enter the complete path to the `imqservlet.jar` in the Servlet Classpath field. For example:

   `/usr/share/lib/imq/imqservlet.jar` (on Solaris)

   `/opt/imq/lib/imqservlet.jar` (on Linux)

   `IMQ_HOME/lib/imqservlet.jar` (on Windows)

6. In the Servlet args field, enter any optional arguments, as shown in Table C-2:

Table C-2 Servlet Arguments for Deploying HTTP Tunnel Servlet Jar File

Argument	Default Value	Reference
servletHost	all hosts	See Table C-1 on page 309
servletPort	7675	See Table C-1 on page 309

If using both arguments, separate them with a comma:

`servletPort=`*portNumber*`, servletHost=...`

The `servletHost` and `servletPort` argument apply only to communication between the Web Server and broker, and are set only if the default values are problematic. However, in that case, you also have to set the broker configuration properties accordingly (see Table C-1 on page 309), for example:

```
imq.httpjms.http.servletPort
```

Configuring a Servlet Virtual Path (Servlet URL)

➤ **To Configure a Virtual Path (Servlet URL) for a Tunnel Servlet**

1. Select the Servlets tab.
2. Choose Configure Servlet Virtual Path Translation.
3. Set the Virtual Path field.

   The Virtual Path is the */contextRoot*/`tunnel` portion of the tunnel servlet URL:

   `http://`*hostName*`:`*port*`/`*contextRoot*`/tunnel`

   For example, if you set the *contextRoot* to `imq`, then the Virtual Path field would be:

   ```
   /imq/tunnel
   ```

4. Set the Servlet Name field to the same value as in Step 3 in "Adding a Servlet" on page 312.

Loading a Servlet

➤ **To Load the Tunnel Servlet at Web Server Startup**

1. Select the Servlets tab.
2. Choose Configure Global Attributes.
3. In the Startup Servlets field, enter the same servlet name value as in Step 3 in "Adding a Servlet" on page 312.

Disabling a Server Access Log

You do not have to disable the server access log, but you will obtain better performance if you do.

➤ **To Disable the Server Access Log**

1. Select the Status tab.
2. Choose the Log Preferences Page.
3. Use the Log client accesses control to disable logging

Deploying as a WAR File

The instructions below refer to deployment on Sun ONE Web Server 6.0 Service Pack 2. You can verify successful HTTP tunnel servlet deployment by accessing the servlet URL using a web browser. It should display status information.

➤ **To Deploy the http Tunnel Servlet as a WAR File**

1. In the browser-based administration GUI, select the Virtual Server Class tab and select Manage Classes.
2. Select the appropriate virtual server class name (for example, `defaultClass`) and click the Manage button.
3. Select Manage Virtual Servers.
4. Select an appropriate virtual server name and click the Manage button.
5. Select the Web Applications tab.
6. Click on Deploy Web Application.
7. Select the appropriate values for the WAR File On and WAR File Path fields so as to point to the `imqhttp.war` file, which can be found in a directory that depends on your operating system (see Appendix A, "Location of MQ Data").

8. Enter a path in the Application URI field.

   The Application URI field value is the */contextRoot* portion of the tunnel servlet URL:

   `http://`*hostName*`:`*port*`/`*contextRoot*`/tunnel`

   For example, if you set the *contextRoot* to `imq`, then the Application URI field would be:

   `/imq`

9. Enter the installation directory path (typically somewhere under the Sun ONE Web Server installation root) where the servlet should be deployed.
10. Click OK.
11. Restart the web server instance.

The servlet is now available at the following address:

`http://`*hostName*`:`*port*`/`*contextRoot*`/tunnel`

Clients can now use this URL to connect to the message service using an HTTP connection.

Example 2: Deploying the HTTP Tunnel Servlet on Sun ONE Application Server 7.0

This section describes how you deploy the HTTP tunnel servlet as a WAR file on the Sun ONE Application Server 7.0.

Two steps are required:

- deploy the HTTP tunnel servlet using the Application Server 7.0 deployment tool
- modify the application server instance's `server.policy` file

Using the Deployment Tool

➤ **To Deploy the HTTP Tunnel Servlet in an Application Server 7.0 Environment**

1. In the web-based administration GUI, choose

   App Server > Instances > server1 > Applications > Web Applications.

2. Click the Deploy button.

3. In the File Path: textfield, enter the location of the HTTP tunnel servlet WAR file (`imqhttp.war`).

   The location of the `imqhttp.war` file depends on your operating system (see Appendix A, "Location of MQ Data")

4. Click OK.

5. On the next screen, set the value for the Context Root textfield.

   The Context Root field value is the */contextRoot* portion of the tunnel servlet URL:

   `http://`*hostName*`:`*port*`/`*contextRoot*`/tunnel`

   For example, you could set the Context Root field to:

   ```
   /imq
   ```

6. Click OK.

   The next screen shows that the tunnel servlet has been successfully deployed, is enabled by default, and—in this case—is located at:

   ```
   /var/opt/SUNWappserver7/domains/domain1/server1/applications/
   j2ee-modules/imqhttp_1
   ```

The servlet is now available at the following address:

`http://`*hostName*`:`*port*`/`*contextRoot*`/tunnel`

Clients can now use this URL to connect to the message service using an HTTP connection.

Modifying the server.policy File

The Application Server 7.0 enforces a set of default security policies that unless modified would prevent the HTTP tunnel servlet from accepting connections from the MQ broker.

Each application server instance has a file that contains its security policies or rules. For example, the location of this file for the server1 instance on Solaris is:

```
/var/opt/SUNWappserver7/domains/domain1/server1/config/
server.policy
```

To make the tunnel servlet accept connections from the MQ broker, an additional entry is required in this file.

➤ **To Modify the Application Server's server.policy File**

1. Open the server.policy file.
2. Add the following entry:

```
grant codeBase
"jar:file:/var/opt/SUNWappserver7/domains/domain1/server1/
            applications/j2ee-modules/imqhttp_1/-"
{
    permission java.net.SocketPermission "*",
            "connect,accept,resolve";
};
```

Enabling HTTPS Support

The following sections describe the steps you need to take to enable HTTPS support. They are similar to those in "Enabling HTTP Support" on page 307 with the addition of steps needed to generate and access SSL certificates.

➤ **To Enable HTTPS Support**

1. Generate a self-signed certificate for the HTTPS tunnel servlet.
2. Deploy the HTTPS tunnel servlet on a web server.
3. Configure the broker's httpsjms connection service and start the broker.
4. Configure an HTTPS connection.

Each of these steps is discussed in more detail in the sections that follow.

Step 1. Generating a Self-signed Certificate for the HTTPS Tunnel Servlet

MQ's SSL support is oriented toward securing on-the-wire data with the assumption that the client is communicating with a known and trusted server. Therefore, SSL is implemented using only self-signed server certificates. In the httpsjms connection service architecture, the HTTPS tunnel servlet plays the role of server to both broker and application client.

Run the `imqkeytool` utility to generate a self-signed certificate for the tunnel servlet. Enter the following at the command prompt:

```
imqkeytool -servlet keystore_location
```

The utility will prompt you for the information it needs. (On Unix systems you may need to run `imqkeytool` as the superuser (root) in order to have permission to create the keystore.)

First, `imqkeytool` prompts you for a keystore password, then it prompts you for some organizational information, and then it prompts you for confirmation. After it receives the confirmation, it pauses while it generates a key pair. It then asks you for a password to lock the particular key pair (key password); you should enter Return in response to this prompt: this makes the key password the same as the keystore password.

NOTE Remember the password you provide—you will need to provide this password later to the tunnel servlet so it can open the keystore.

Running `imqkeytool` runs the JDK `keytool` utility to generate a self-signed certificate and to place it in MQ's keystore file located as specified in the *keystore_location* argument. (The keystore is in the same keystore format as that supported by the JDK1.2 `keytool`.)

NOTE The HTTPS tunnel servlet must be able to see the keystore. Make sure you move/copy the generated keystore located in *keystore_location* to a location accessible by the HTTPS tunnel servlet (see "Step 2. Deploying the HTTPS Tunnel Servlet on a Web Server" on page 318).

Step 2. Deploying the HTTPS Tunnel Servlet on a Web Server

There are two general ways you can deploy the HTTPS tunnel servlet on a web server:

- deploying it as a jar file—for web servers that support Servlet 2.1 or earlier
- deploying it as a web archive (WAR) file—for web servers that support Servlet 2.2 or later

In either case, you should make sure that encryption is activated for the web server, enabling end to end secure communication between the client and broker.

Deploying as a Jar File

Deploying the MQ tunnel servlet consists of making the appropriate jar files accessible to the host web server, configuring the web server to load the servlet on startup, and specifying the context root portion of the servlet's URL.

The tunnel servlet jar file (`imqservlet.jar`) contains all the classes needed by the HTTPS tunnel servlet, and can be found in a directory that depends upon operating system (see Appendix A, "Location of MQ Data").

Any web server with servlet 2.x support can be used to load this servlet. The servlet class name is:

```
com.sun.messaging.jmq.transport.
httptunnel.servlet.HttpsTunnelServlet
```

The web server must be able to see the `imqservlet.jar` file. If you are planning to run the web server and the broker on different hosts, you should place a copy of the `imqservlet.jar` file in a location where the web server can access it.

You also need to configure the web server to load this servlet on startup, and you might need to specify the context root portion of the servlet's URL (see "Example 3: Deploying the HTTPS Tunnel Servlet on Sun ONE Web Server" on page 324).

Make sure that the JSSE jar files are in the classpath for running servlets in the web server. Check the web server's documentation for how to do this.

An important aspect of configuring the web server is specifying the location and password of the self-signed certificate to be used by the HTTPS tunnel servlet to establish a secure connection with a broker. You have to place the keystore created in "Step 1. Generating a Self-signed Certificate for the HTTPS Tunnel Servlet" on page 317 in a location accessible by the HTTPS tunnel servlet.

It is also recommended that you disable your web server's access logging feature in order to improve performance.

Deploying as a Web Archive File

Deploying the HTTPS tunnel servlet as a WAR file consists of using the deployment mechanism provided by the web server. The HTTPS tunnel servlet WAR file (`imqhttps.war`) is located in a directory that depends on your operating system (see Appendix A, "Location of MQ Data").

The WAR file includes a deployment descriptor that contains the basic configuration information needed by the web server to load and run the servlet. Depending on the web server, you might also need to specify the context root portion of the servlet's URL (see "Example 4: Deploying the HTTPS Tunnel Servlet on Sun ONE Application Server 7.0" on page 329).

However, the deployment descriptor of the `imqhttps.war` file cannot know where you have placed the keystore file needed by the tunnel servlet (see "Step 1. Generating a Self-signed Certificate for the HTTPS Tunnel Servlet" on page 317). This requires you to edit the tunnel servlet's deployment descriptor (an XML file) to specify the keystore location before deploying the `imqhttps.war` file.

Step 3. Configuring the httpsjms Connection Service

HTTPS support is not activated for a broker by default, so you need to reconfigure the broker to activate the httpsjms connection service. Once reconfigured, the broker can be started as outlined in "Starting a Broker" on page 134.

➤ **To Activate the httpsjms Connection Service**

1. Open the broker's instance configuration file.

   The instance configuration file is stored in a directory identified by the name of the broker instance (*instanceName*) with which the configuration file is associated (see Appendix A, "Location of MQ Data"):

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

2. Add the `httpsjms` value to the `imq.service.activelist` property:

   ```
   imq.service.activelist=jms,admin,httpsjms
   ```

At startup, the broker looks for a web server and HTTPS tunnel servlet running on its host machine. To access a remote tunnel servlet, however, you can reconfigure the servletHost and servletPort connection service properties.

You can also reconfigure the pullPeriod property to improve performance. The httpsjms connection service configuration properties are detailed in Table C-3.

Table C-3 httpsjms Connection Service Properties

Property Name	Description
`imq.httpsjms.https.servletHost`	Change this value, if necessary, to specify the name of the host (hostname or IP address) on which the HTTPS tunnel servlet is running. (This can be a remote host or a specific hostname on a local host.) Default: `localhost`
`imq.httpsjms.https.servletPort`	Change this value to specify the port number that the broker uses to access the HTTPS tunnel servlet. (If the default port is changed on the Web server, then you must change this property accordingly.) Default: `7674`
`imq.httpsjms.https.pullPeriod`	Specifies the interval, in seconds, between HTTP requests made by each client to pull messages from the broker. (Note that this property is set on the broker and propagates to the client runtime.) If the value is zero or negative, the client keeps one HTTP request pending at all times, ready to pull messages as fast as possible. With a large number of clients, this can be a heavy drain on web server resources and the server may become unresponsive. In such cases, you should set the `pullPeriod` property to a positive number of seconds. This sets the time the client's HTTP transport driver waits before making subsequent pull requests. Setting the value to a positive number conserves web server resources at the expense of the response times observed by clients. Default: `-1`
`imq.httpsjms.https.connectionTimeout`	Specifies the time, in seconds, that the client runtime waits for a response from the HTTPS tunnel servlet before throwing an exception. (Note that this property is set on the broker and propagates to the client runtime.) This property also specifies the time the broker waits after communicating with the HTTPS tunnel servlet before freeing up a connection. A timeout is necessary in this case because the broker and the tunnel servlet have no way of knowing if a client that is accessing the HTTPS servlet has terminated abnormally. Default: `60`

Step 4. Configuring an HTTPS Connection

A client application must use an appropriately configured connection factory administered object to make an HTTPS connection to a broker.

However, the client must also have access to SSL libraries provided by the Java Secure Socket Extension (JSSE) and must also have a root certificate. The SSL libraries are bundled with JDK 1.4. If you have an earlier JDK version, see "Configuring JSSE," otherwise proceed to "Importing a Root Certificate."

Once these issues are resolved, you can proceed to configuring the HTTPS connection.

Configuring JSSE

➤ **To Configure JSSE**

1. Copy the JSSE jar files to the `JRE_HOME/lib/ext` directory.

   `jsse.jar, jnet.jar, jcert.jar`

2. Statically add the JSSE security provider by adding

   `security.provider.`*n*`=com.sun.net.ssl.internal.ssl.Provider`

   to the `JRE_HOME/lib/security/java.security` file (where *n* is the next available priority number for security provider package).

3. If not using JDK1.4, you need to set the following JSSE property using the `-D` option to the command that launches the client application:

   `java.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol`

Importing a Root Certificate

If the root certificate of the CA who signed your web server's certificate is not in the trust database by default or if you are using a proprietary web server certificate, you have to add that certificate to the trust database. If this is the case, follow the instruction below, otherwise go to "Configuring the Connection Factory."

Assuming that the certificate is saved in *cert_file* and that *trust_store_file* is your keystore, run the following command:

```
JRE_HOME/bin/keytool -import -trustcacerts
-alias alias_for_certificate -file cert_file
-keystore trust_store_file
```

Answer `YES` to the question: Trust this certificate?

You also need to specify the following JSSE properties using the `-D` option to the command that launches the client application:

`javax.net.ssl.trustStore=`*trust_store_file*

`javax.net.ssl.trustStorePassword=`*trust_store_passwd*

Configuring the Connection Factory

To enable HTTPS support, you need to set the connection factory's `imqAddressList` attribute to the HTTPS tunnel servlet URL. The general syntax of the HTTPS tunnel servlet URL is the following:

`https://`*hostName:port/contextRoot*`/tunnel`

where *hostName:port* is the name and port of the web server hosting the HTTPS tunnel servlet and *contextRoot* is a path set when deploying the tunnel servlet on the web server.

For more information on connection factory attributes in general, and the `imqAddressList` attribute in particular, see the MQ *Java Client Developer's Guide*.

You can set connection factory attributes in one of the following ways:

- Using the `-o` option to the `imqobjmgr` command that creates the connection factory administered object (see "Adding a Connection Factory" on page 195), or set the attribute when creating the connection factory administered object using the Administration Console (`imqadmin`).
- Using the `-D` option to the command that launches the client application (see the MQ *Java Client Developer's Guide*).
- Using an API call to set the attributes of a connection factory after you create it programmatically in client application code (see the MQ *Java Client Developer's Guide*).

Using a Single Servlet to Access Multiple Brokers

You do not need to configure multiple web servers and servlet instances if you are running multiple brokers. You can share a single web server and HTTPS tunnel servlet instance among concurrently running brokers. If multiple broker instances are sharing a single tunnel servlet, you must configure the `imqAddressList` connection factory attribute as shown below:

`https://`*hostName:port/contextRoot*`/tunnel?ServerName=`*bkrHostName*`:`*instanceName*

Where *bkrHostName* is the broker instance host name and *instanceName* is the name of the specific broker instance you want your client to access.

To check that you have entered the correct strings for *bkrhostName* and *instanceName*, generate a status report for the HTTPS tunnel servlet by accessing the servlet URL from a browser. The report lists all brokers being accessed by the servlet:

```
HTTPS tunnel servlet ready.
Servlet Start Time : Thu May 30 01:08:18 PDT 2002
Accepting secured connections from brokers on port : 7674
Total available brokers = 2
Broker List :
   jpgserv:broker2
   cochin:broker1
```

Using an HTTP Proxy

If you are using an HTTP proxy to access the HTTPS tunnel servlet:

- Set `http.proxyHost` system property to the proxy server host name.
- Set `http.proxyPort` system property to the proxy server port number.

You can set these properties using the `-D` option to the command that launches the client application.

Example 3: Deploying the HTTPS Tunnel Servlet on Sun ONE Web Server

This section describes how you deploy the HTTPS tunnel servlet both as a jar file and as a WAR file on the Sun ONE Web Server. The approach you use depends on the version of Sun ONE Web Server: If it does not support Servlet 2.2 or later, it will not be able to handle WAR file deployment.

Deploying as a Jar File

The instructions below refer to deployment on Sun ONE Web Server 6.1 using the browser-based administration GUI. This procedure consists of the following general steps:

1. add a servlet
2. configure the servlet virtual path
3. load the servlet
4. disable the servlet access log

These steps are described in the following subsections. You can verify successful HTTPS tunnel servlet deployment by accessing the servlet URL using a web browser. It should display status information.

Adding a Servlet

➤ **To Add a Tunnel Servlet**

1. Select the Servlets tab.
2. Choose Configure Servlet Attributes.
3. Specify a name for the tunnel servlet in the Servlet Name field.
4. Set the Servlet Code (class name) field to the following value:

   ```
   com.sun.messaging.jmq.transport.
   httptunnel.servlet.HttpsTunnelServlet
   ```

5. Enter the complete path to the `imqservlet.jar` in the Servlet Classpath field. For example:

   `/usr/share/lib/imq/imqservlet.jar` (on Solaris)

   `/opt/imq/lib/imqservlet.jar` (on Linux)

   `IMQ_HOME/lib/imqservlet.jar` (on Windows)

6. In the Servlet args field, enter required and optional arguments, as shown in Table C-4.

Table C-4 Servlet Arguments for Deploying HTTPS Tunnel Servlet Jar File

Argument	Default Value	Required ?	See Also
`keystoreLocation`	none	Yes	Table 8-8 on page 220
`keystorePassword`	none	Yes	Table 8-8 on page 220
`servletHost`	all hosts	No	Table C-3 on page 321
`servletPort`	7674	No	Table C-3 on page 321

Separate the arguments with a comma, for example:

`keystoreLocation=`*keystore_location*`,keystorePassword=`*keystore_password*`, servletPort=`*portnumber*

The `servletHost` and `servletPort` argument apply only to communication between the Web Server and broker, and are set only if the default values are problematic. However, in that case, you also have to set the broker configuration properties accordingly (see Table C-3 on page 321), for example:

```
imq.httpsjms.https.servletPort
```

Configuring a Servlet Virtual Path (Servlet URL)

➤ **To Configure a Virtual Path (servlet URL) for a Tunnel Servlet**

1. Select the Servlets tab.
2. Choose Configure Servlet Virtual Path Translation.
3. Set the Virtual Path field.

   The Virtual Path is the */contextRoot*/`tunnel` portion of the tunnel servlet URL:

   `https://`*hostName*`:`*port*`/`*contextRoot*`/tunnel`

   For example, if you set the *contextRoot* to `imq`, then the Virtual Path field would be:

   ```
   /imq/tunnel
   ```

4. Set the Servlet Name field to the same value as in Step 3 in "Adding a Servlet" on page 325.

Loading a Servlet

➤ **To Load the Tunnel Servlet at Web Server Startup**

1. Select the Servlets tab.
2. Choose Configure Global Attributes.
3. In the Startup Servlets field, enter the same servlet name value as in Step 3 in "Adding a Servlet" on page 325.

Disabling a Server Access Log

You do not have to disable the server access log, but you will obtain better performance if you do.

➤ **To Disable the Server Access Log**

1. Select the Status tab.
2. Choose the Log Preferences Page.
3. Use the Log client accesses control to disable logging

Deploying as a WAR File

The instructions below refer to deployment on Sun ONE Web Server 6.0 Service Pack 2. You can verify successful HTTPS tunnel servlet deployment by accessing the servlet URL using a web browser. It should display status information.

Before deploying the HTTPS tunnel servlet, make sure that JSSE jar files are included in the web server's classpath. The simplest way to do this is to copy the `jsse.jar`, `jnet.jar`, and `jcert.jar` to `IWS60_TOPDIR/bin/https/jre/lib/ext`.

Also, before deploying the HTTPS tunnel servlet, you have to modify its deployment descriptor to point to the location where you have placed the keystore file and to specify the keystore password.

➤ **To Modify the HTTPS Tunnel Servlet WAR File**

1. Copy the WAR file to a temporary directory.

   `cp /usr/share/lib/imq/imqhttps.war /tmp` (on Solaris)

   `cp /opt/imq/lib/imqhttps.war /tmp` (on Linux)

   `cp IMQ_HOME/lib/imqhttps.war /tmp` (on Windows)

2. Make the temporary directory your current directory.

   ```
   $ cd /tmp
   ```

3. Extract the contents of the WAR file.

   ```
   $ jar xvf imqhttps.war
   ```

4. List the WAR file's deployment descriptor.

   ```
   $ ls -l WEB-INF/web.xml
   ```

5. Edit the `web.xml` file to provide correct values for the `keystoreLocation` and `keystorePassword` arguments (as well as `servletPort` and `servletHost` arguments, if necessary).

6. Re-assemble the contents of the WAR file.

   $ jar uvf imqhttps.war WEB-INF/web.xml

You are now ready to use the modified `imqhttps.war` file to deploy the HTTPS tunnel servlet. (If you are concerned about exposure of the keystore password, you can use file system permissions to restrict access to the `imqhttps.war` file.)

➤ To Deploy the https Tunnel Servlet as a WAR File

1. In the browser-based administration GUI, select the Virtual Server Class tab. Click Manage Classes.
2. Select the appropriate virtual server class name (for example, `defaultClass`) and click the Manage button.
3. Select Manage Virtual Servers.
4. Select an appropriate virtual server name and click the Manage button.
5. Select the Web Applications tab.
6. Click on Deploy Web Application.
7. Select the appropriate values for the WAR File On and WAR File Path fields so as to point to the modified `imqhttps.war` file (see "To Modify the HTTPS Tunnel Servlet WAR File" on page 327.)
8. Enter a path in the Application URI field.

   The Application URI field value is the */contextRoot* portion of the tunnel servlet URL:

   `https://`*hostName*`:`*port*`/`*contextRoot*`/tunnel`

   For example, if you set the *contextRoot* to `imq`, then the Application URI field would be:

   `/imq`
9. Enter the installation directory path (typically somewhere under the Sun ONE Web Server installation root) where the servlet should be deployed.
10. Click OK.
11. Restart the web server instance.

The servlet is now available at the following address:

`https://`*hostName*`:`*port*`/imq/tunnel`

Clients can now use this URL to connect to the message service using a secure HTTPS connection.

Example 4: Deploying the HTTPS Tunnel Servlet on Sun ONE Application Server 7.0

This section describes how you deploy the HTTPS tunnel servlet as a WAR file on the Sun ONE Application Server 7.0.

Two steps are required:

- deploy the HTTPS tunnel servlet using the Application Server 7.0 deployment tool
- modify the application server instance's `server.policy` file

Using the Deployment Tool

➤ **To Deploy the HTTPS Tunnel Servlet in an Application Server 7.0 Environment**

1. In the web-based administration GUI, choose

   App Server > Instances > server1 > Applications > Web Applications.

2. Click the Deploy button.
3. In the File Path: textfield, enter the location of the HTTPS tunnel servlet WAR file (`imqhttps.war`).

   The location of the `imqhttps.war` file depends on your operating system (see Appendix A, "Location of MQ Data")

4. Click OK.
5. On the next screen, set the value for the Context Root textfield.

   The Context Root field value is the */contextRoot* portion of the tunnel servlet URL:

   `https://`*hostName*`:`*port*`/`*contextRoot*`/tunnel`

   For example, you could set the Context Root field to:

   `/imq`

6. Click OK.

   The next screen shows that the tunnel servlet has been successfully deployed, is enabled by default, and—in this case—is located at:

   ```
   /var/opt/SUNWappserver7/domains/domain1/server1/applications/
   j2ee-modules/imqhttps_1
   ```

The servlet is now available at the following address:

`https://`*hostName*`:`*port*`/`*contextRoot*`/tunnel`

Clients can now use this URL to connect to the message service using an HTTPS connection.

Modifying the server.policy file

The Application Server 7.0 enforces a set of default security policies that unless modified would prevent the HTTPS tunnel servlet from accepting connections from the MQ broker.

Each application server instance has a file that contains its security policies or rules. For example, the location of this file for the server1 instance on Solaris is:

```
/var/opt/SUNWappserver7/domains/domain1/server1/config/
server.policy
```

To make the tunnel servlet accept connections from the MQ broker, an additional entry is required in this file.

➤ **To Modify the Application Server's server.policy File**

1. Open the server.policy file.
2. Add the following entry:

```
grant codeBase
"jar:file:/var/opt/SUNWappserver7/domains/domain1/server1/
            applications/j2ee-modules/imqhttps_1/-”
{
    permission java.net.SocketPermission "*",
            “connect,accept,resolve";
};
```

Appendix D

Using a Broker as a Windows Service

This appendix explains how you use the Service Administrator (`imqsvcadmin`) utility to install, query, and remove a broker running as a Windows Service.

Running a Broker as a Windows Service

You have the option of installing a broker as a Windows service when you install Sun™ ONE Message Queue. You can also use `imqsvcadmin` to install a broker as a Windows service after you have installed MQ.

Installing a broker as a Windows service means that it will start at system startup time and run in the background until you shut down. Consequently, you do not use the `imqbrokerd` command to start the broker—unless, you want to start an additional instance. To pass any start-up options to the broker, you can use the `-args` argument to the `imqsvcadmin` command (see Table D-2 on page 333) and specify exactly the same options you would have used for the `imqbrokerd` command (see "Starting a Broker" on page 134). Use the `imqcmd` command to control broker operations as usual.

When running as a Windows service, the Task Manager lists the broker as two executable processes. The first is `imqbrokersvc.exe`, which is the native Windows service wrapper. The second is the Java runtime that is actually running the broker.

Only one broker at a time can be installed and run as a Windows service.

Service Administrator Utility (imqsvcadmin)

The Service Administrator utility (`imqsvcadmin`) allows you to install, query, and remove the broker (running as a Windows service). This section describes the basic syntax of `imqsvcadmin` commands, provides a listing of subcommands, summarizes `imqsvcadmin` command options, and explains how to use these commands to accomplish specific tasks.

Syntax of the imqsvcadmin Command

The general syntax of `imqsvcadmin` commands is as follows:

```
imqsvcadmin subcommand [options]
```

```
imqsvcadmin -h
```

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

```
imqsvcadmin query -h
```

imqsvcadmin Subcommands

The MQ Service Administrator utility (`imqsvcadmin`) includes the subcommands listed in Table D-1:

Table D-1 `imqsvcadmin` Subcommands

Subcommand	Description
`install`	Installs the service and specifies startup options.
`query`	Displays the startup options to the `imqsvcadmin` command. This includes whether the service is started manually or automatically, its location, the location of the java runtime, and the value of the arguments passed to the broker on startup.
`remove`	Removes the service.

Summary of imqsvcadmin Options

Table D-2 lists the options to the `imqsvcadmin` command. For a discussion of their use, see the task-based sections that follow.

Table D-2 `imqsvcadmin` Options

Option	Description
`-h`	Displays usage help. Nothing else on the command line is executed.
`-javahome` *path*	Specifies the path to an alternate Java 2 compatible runtime to use (default is to use the runtime on the system or the runtime bundled with MQ. Example: `imqsvcadmin -install -javahome d:\jdk1.4`
`-jrehome` *path*	Specifies the path to a Java 2 compatible JRE. Example: `imqsvcadmin -install -jrehome d:\jre\1.4`
`-vmargs` *arg* [[*arg*]…]	Specifies additional arguments to pass to the Java VM that is running the broker service. (You can also specify these arguments in the Windows Services Control Panel Startup Parameters field.) Example: `-vmargs "-Xms16m -Xmx128m"`
`-args` *arg* [[*arg*]…]	Specifies additional command line arguments to pass to the broker service. For a description of the `imqbrokerd` options, see "Starting a Broker" on page 134. (You can also specify these arguments in the Windows Services Control Panel Startup Parameters field.) For example, `imqsvcadmin -install -args "-passfile d:\imqpassfile"`

The information that you specify using the `-javahome`, `-vmargs`, and `-args` options is stored in the Window's registry under the keys `JavaHome`, `JVMArgs`, and `ServiceArgs` in the path

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet
     \Services\iMQ_Broker\Parameters
```

Removing the Broker Service

Before you remove the broker service, you should use the `imqcmd shutdown bkr` command to shut down the broker. Then use the `imqsvcadmin remove` command to remove the service, and restart your computer.

Reconfiguring the Broker Service

To reconfigure the service, remove the service first, and then reinstall it, specifying different startup options with the `-args` argument.

Using an Alternate Java Runtime

You can use either the `-javahome` or `-jrehome` options to specify the location of an alternate java runtime. You can also specify these options in the Windows Services Control Panel Startup Parameters field. Note that the Startup Parameters field treats the back slash (\) as an escape character, so you will have to type it twice when using it as a path delimiter; for example, `-javahome d:\\jdk1.3`.

Querying the Broker Service

To determine the startup options for the broker service, use the `-q` option to the `imqsvcadmin` command.

```
imqsvcadmin -query

Service iMQ_Broker is installed.
Display Name: iMQ_Broker
Start Type: Manual
Binary location: c:\Program Files\Sun Microsystems\
                    Message Queue 3.5\bin\imqbrokersvc
JavaHome: c:\j2sdk1.4.0
Broker Args: -passfile d:\imqpassfile
```

Troubleshooting

If you get an error when you try and start the service, you can see error events that were logged by doing the following.

➤ **To See Logged Service Error Events**

1. Start the Event Viewer
2. Look under Log > Application.
3. Select View > Refresh to see any error events.

Appendix E

The MQ Resource Adapter

Message Queue includes a JMS resource adapter.

A resource adapter is a standardized way for plugging additional functionality into a J2EE 1.4 compliant application server, in compliance with the J2EE Connector Architecture (J2EECA) 1.5 specification. This architecture allows any J2EE 1.4 compliant application server to interact with external systems in a standardized way. These external systems include various enterprise information systems (EIS), as well as various messaging systems, for example, a JMS provider.

The standardized interactions facilitated by J2EECA 1.5 include connection pooling, thread pooling, transaction and security context propagation, as well as support for message driven bean containers of various kinds. The specification also includes a standardized way to create connection factories and other administered objects.

By plugging a JMS resource adapter into an application server, J2EE components deployed and running in the application server environment can exchange JMS messages. The JMS connection factory and destination administered objects needed by these components can be created and configured using J2EE application server administration tools.

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

The MQ resource adapter is embedded in the Sun J2EE 1.4 Application Server. The MQ resource adapter, however has not yet been certified with any other J2EE 1.4 application server.

The MQ resource adapter is a single file (`imqjmsra.rar`) and is located in a directory that depends on operating system, as shown in Appendix A, "Location of MQ Data." The `imqjmsra.rar` file contains the resource adapter deployment descriptor (`ra.xml`) as well as the jar files that must be used by the application server in order to use the adapter.

You can use the MQ resource adapter in any J2EE 1.4 compliant application server by following the resource adapter deployment and configuration instructions that come with that application server. As commercial J2EE 1.4 application servers become available, and the MQ resource adapter becomes certified for those application servers, this appendix will provide specific information on the relevant deployment and configuration procedures.

Appendix F

Technical Notes

This appendix contains short write-ups on the following topics:

- System Clock Settings
- OS-Defined File Descriptor Limitations
- Securing Persistent Data

System Clock Settings

When using an MQ system, you should be careful to synchronize system clocks and avoid setting them backward.

Synchronization Recommended

It is recommended that you synchronize the clocks on all hosts interacting with the MQ system. This is particularly important if you are using message expiration (TimeToLive). Failure to synchronize the hosts' clocks may result in TimeToLive not working as expected (messages may not be delivered). You should synchronize clocks before starting any brokers.

Solaris You can issue the `rdate` command on a local host to synchronize with remote host. (You must be superuser--that is, root--to run this command.) For example, the following command synchronizes the local host (call it Host 2) with remote host Host1:

```
# rdate Host1
```

Linux The command is similar to Solaris, but you must provide the -s option:

```
# rdate -s Host1
```

Windows you can issue the net command with the time subcommand to synchronize your local host with a remote host. For example, the following command synchronizes the local host (call it Host 2) with remote host Host1:

```
net time \\Host1 /set
```

Avoid Setting System Clocks Backwards

You should avoid setting the system clock backwards on systems running an MQ broker. MQ uses timestamps to help identify internal objects such as transactions and durable subscriptions. If the system clock is set backwards it is theoretically possible that a duplicate internal identifier can be generated. The broker attempts to compensate for this by introducing some randomness to identifiers and by detecting clock shift when running, but if the system clock is shifted backwards by a significant amount when a broker is not running, then there is a slight risk of identifier duplication.

If you need to set the system clock backwards on a system running a broker by more than a few seconds, it is recommended that you either do it when there are no transactions or durable subscriptions, or do it when the broker is not running, then wait the amount of time you have shifted the clock before bringing the broker back up.

But the ideal approach is to synchronize clocks before starting up any brokers, and then use an appropriate technique to ensure that clocks don't drift significantly after deployment.

OS-Defined File Descriptor Limitations

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

As a result, the number of connections is limited by these factors. You cannot have a broker or client running with more than 256 connections on Solaris or 1024 on Linux without changing the file descriptor limit. (The connection limit is actually lower than that due to the use of file descriptors for persistence.)

To change the file descriptor limit, see the `ulimit` man page. The limit needs to be changed in each shell in which a client or broker will be executing.

Securing Persistent Data

The broker uses a persistent store that can contain, among other information, message files that are being temporarily stored. Since these messages might contain proprietary information, it is recommended that the data store be secured against unauthorized access.

A broker can use either the built-in or plugged-in persistence.

Built-in Persistent Store

A broker using built-in persistence writes persistent data to a flat file data store located in a directory that depends upon platform (see Appendix A, "Location of MQ Data"):

…/`instances/`*instanceName*`/fs350/`

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

The *instanceName*`/filestore/` directory is created when the broker instance is started for the first time. The procedure for securing this directory depends on the operating system on which the broker is running.

Solaris and Linux The permissions on the `IMQ_VARHOME/instances/`*instanceName*`/filestore/` directory depend on the umask of the user that started the broker instance. Hence, permission to start a broker instance and to read its persistent files can be restricted by appropriately setting the umask. Alternatively, an administrator (superuser) can secure persistent data by setting the permissions on the `IMQ_VARHOME/instances` directory to 700.

Windows The permissions on the `IMQ_VARHOME/instances/`*instanceName*`/filestore/` directory can be set using the mechanisms provided by the Windows operating system that you are using. This generally involves opening a properties dialog for the directory.

Plugged-in Persistent Store

A broker using plugged-in persistence writes persistent data to a JDBC Compliant database.

For a database managed by a database server (for example, an Oracle database), it is recommended that you create a user name and password to access the MQ database tables (tables whose names start with "IMQ"). If the database does not allow individual tables to be protected, create a dedicated database to be used only by MQ brokers. See the database vendor for documentation on how to create user name/password access.

The user name and password required to open a database connection by a broker can be provided as broker configuration properties. However it is more secure to provide them as command line options when starting up the broker (see MQ *Administrator's Guide*, Appendix A, "Setting Up Plugged-in Persistence").

For an embedded database that is accessed directly by the broker via the database's JDBC™ driver (for example, a Cloudscape database), security is usually provided by setting file permissions (as described in "Built-in Persistent Store," above) on the directory where the persistent data will be stored. To ensure that the database is readable and writable by both the broker and the `imqdbmgr` utility, however, both should be run by the same user.

Appendix G

Stability of MQ Interfaces

Sun™ ONE Message Queue (MQ) uses many interfaces, that might be of use to administrators for automating administration tasks. Table G-1 classifies these interfaces according to how stable they are, that is, how unlikely they are to change in subsequent versions of the product. The classification scheme is described in Table G-2 on page 343.

Table G-1 Stability of MQ Interfaces

Interface	Classification
`imqbrokerd` command line interface	Evolving
`imqadmin` command line interface	Unstable
`imqcmd` command line interface	Evolving
`imqdbmgr` command line interface	Unstable
`imqkeytool` command line interface	Evolving
`imqobjmgr` command line interface	Evolving
`imqusermgr` command line interface	Unstable
`imqobjmgr` command file	Evolving
`imqbrokerd` command	Stable
`imqadmin` command	Unstable
`imqcmd` command	Stable
`imqdbmgr` command	Unstable
`imqkeytool` command	Stable
`imqobjmgr` command	Stable
`imqusermgr` command	Unstable

Table G-1 Stability of MQ Interfaces *(Continued)*

Interface	Classification
JMS API (`javax.jms`)	Standard
JAXM API (`javax.xml`)	Standard
C-API	Evolving
Message-based monitoring API	Evolving
Administered Object API (`com.sun.messaging`)	Evolving
`imq.jar` location and name	Stable
`jms.jar` location and name	Evolving
`imqbroker.jar` location and name	Private
`imqutil.jar` location and name	Private
`imqadmin.jar` location and name	Private
`imqservlet.jar` location and name	Evolving
`imqhttp.war` location and name	Evolving
`imqhttps.war` location and name	Evolving
`imqjmsra.rar` location and name	Evolving
`imqxm.jar` location and name	Evolving
`jaxm-api.jar` location and name	Evolving
`saaj-api.jar` location and name	Evolving
`saaj-impl.jar` location and name	Evolving
`activation.jar` location and name	Evolving
`mail.jar` location and name	Evolving
`dom4j.jar` location and name	Private
`fscontext.jar` location and name	Unstable
Output from `imqbrokerd`, `imqadmin`, `imqcmd`, `imqdbmgr`, `imqkeytool`, `imqobjmgr`, `imqusermgr`	Unstable
Broker log file location and content format	Unstable
passfile	Unstable
`accesscontrol.properties`	Unstable

Table G-2 Interface Stability Classification Scheme

Classification	Description
Private	Not for direct use by customers. May change or be removed in any release.
Evolving	For use by customers. Subject to incompatible change at a major (e.g. 3.0, 4.0) or minor (e.g. 3.1, 3.2) release. The changes will be made carefully and slowly. Reasonable efforts will be made to ensure that all changes are compatible but that is not guaranteed.
Stable	For use by customers. Subject to incompatible change at a major (e.g 3.0, 4.0) release only.
Standard	For use by customers. These interfaces are defined by a formal standard, and controlled by a standards organization. Incompatible changes to these interfaces are rare.
Unstable	For use by customers. Subject to incompatible change at a major (e.g. 3.0, 4.0) or minor (e.g. 3.1, 3.2) release. Customers are advised that these interfaces may be removed or changed substantially and in an incompatible way in a future release. It is recommended that customers not create explicit dependencies on unstable interfaces.

Appendix H

MQ Implementation of Optional JMS Functionality

The JMS specification indicates certain items that are optional-- each JMS provider (vendor) chooses whether or not to implement them. The MQ product handling of each of these optional items is indicated below:

Table H-1 Optional JMS Functionality

Section in JMS Specification	Description and MQ Handling
3.4.3 JMSMessageID	"Since message ID's take some effort to create and increase a message's size, some JMS providers may be able to optimize message overhead if they are given a hint that message ID is not used by an application. JMS Message Producer provides a hint to disable message ID." **MQ implementation**: Product does not disable Message ID generation (any setDisableMessageID() call in MessageProducer is ignored). All messages will contain a valid MessageID value.
3.4.12 Overriding Message Header Fields	"JMS does not define specifically how an administrator overrides these header field values. A JMS provider is not required to support this administrative option." **MQ implementation**: The MQ product supports administrative override of the values in message header fields through configuration of connection factory administered objects (see Table 7-3 on page 187).
3.5.9 JMS Defined Properties	"JMS Reserves the 'JMSX' Property name prefix for JMS defined properties." "Unless noted otherwise, support for these properties is optional." **MQ implementation**: The JMSX properties defined by the JMS 1.1 specification are supported in the MQ product (see Table 7-3 on page 187).

Table H-1 Optional JMS Functionality *(Continued)*

Section in JMS Specification	Description and MQ Handling
3.5.10 Provider-specific Properties	"JMS reserves the 'JMS_<vendor_name>' property name prefix for provider-specific properties."
	MQ implementation: The purpose of the provider-specific properties is to provide special features needed to support JMS use with provider-native clients. They should not be used for JMS to JMS messaging. MQ 3.5 does not use provider-specific properties.
4.4.8 Distributed Transactions	"JMS does not require that a provider support distributed transactions."
	MQ implementation: Distributed transactions are supported in this release of the MQ product (see "Distributed Transactions" on page 46).
4.4.9 Multiple Sessions	"For PTP <point-to-point distribution model>, JMS does not specify the semantics of concurrent QueueReceivers for the same queue; however, JMS does not prohibit a provider from supporting this." See section 5.8 of the JMS specification for more information.
	MQ implementation: The MQ implementation supports queue delivery to multiple consumers. For more information, see "Queue Delivery to Multiple Consumers" on page 76.

Glossary

This glossary provides information about terms and concepts you might encounter while using Sun™ ONE Message Queue (MQ).

administered objects A pre-configured MQ object—a connection factory or a destination—created by an administrator for use by one or more JMS clients.

The use of administered objects allows JMS clients to be provider-independent; that is, it isolates them from the proprietary aspects of a provider. These objects are placed in a JNDI name space by an administrator and are accessed by JMS clients using JNDI lookups.

asynchronous communication A mode of communication in which the sender of a message need not wait for the sending method to return before it continues with other work.

authorization The process by which a message service determines whether a user can access message service resources, such as connection services or destinations.

broker The MQ entity that manages message routing, delivery, persistence, security, and logging, and which provides an interface that allows an administrator to monitor and tune performance and resource use.

client An application (or software component) that interacts with other clients using a message service to exchange messages.

client identifier An identifier that associates a connection and its objects with a state maintained by the MQ message server on behalf of the client.

client runtime See MQ client runtime.

cluster Two or more interconnected brokers that work in tandem to provide messaging services.

configuration file One or more text files containing MQ settings that are used to configure a broker. The properties are instance-specific or cluster-related.

connection 1) An active connection to an MQ message server. This can be a queue connection or a topic connection. 2) A factory for sessions that use the connection underlying MQ message server for producing and consuming messages.

connection factory The administered object the client uses to create a connection to MQ message server. This can be a QueueConnectionFactory object or a TopicConnectionFactory object.

consume The receipt of a message taken from a destination by a message consumer.

consumer An object (MessageConsumer) created by a session that is used for receiving messages from a destination. In the point-to-point delivery model, the consumer is a receiver or browser (QueueReceiver or QueueBrowser); in the publish/subscribe delivery model, the consumer is a subscriber (TopicSubscriber).

data store A database where information (durable subscriptions, data about destinations, persistent messages, auditing data) needed by the broker is permanently stored.

delivery mode An indicator of the reliability of messaging: whether messages are guaranteed to be delivered and successfully consumed once and only once (persistent delivery mode) or guaranteed to be delivered at most once (non-persistent delivery mode).

delivery model The model by which messages are delivered: either point-to-point or publish/subscribe. In JMS there are separate programming domains for each, using specific client runtime objects and specific destination types (queue or topic), as well as a unified programming domain.

delivery policy A specification of how a queue is to route messages when more than one message consumer is registered. The policies are: single, failover, and round-robin.

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

domain A set of objects used by JMS clients to program JMS messaging operations. There are two programming domains: one for the point-to-point delivery model and one for the publish/subscribe delivery model.

MQ client runtime Software that provides JMS clients with an interface to the MQ message server. The client runtime supports all operations needed for clients to send messages to destinations and to receive messages from such destinations.

MQ message server Software that provides delivery services for an MQ messaging system, including connections to JMS clients, message routing and delivery, persistence, security, and logging. The message server maintains physical destinations to which JMS clients send messages, and from which the messages are delivered to consuming clients.

JMS (Java Message Service) A standard set of interfaces and semantics that define how a Java client accesses the facilities of a message service. These interfaces provide a standard way for Java programs to create, send, receive, and read messages.

JMS provider A product that implements the JMS interfaces for a messaging system and adds the administrative and control functions needed for a complete product.

message selector A way for a consumer to select messages based on property values (selectors) in JMS message headers. A message service performs message filtering and routing based on criteria placed in message selectors.

message service See MQ message server.

messages Asynchronous requests, reports, or events that are consumed by JMS clients. A message has a header (to which additional fields can be added) and a body. The message header specifies standard fields and optional properties. The message body contains the data that is being transmitted.

messaging A system of asynchronous requests, reports, or events used by enterprise applications that allows loosely coupled applications to transfer information reliably and securely.

point-to-point delivery model Producers address messages to specific queues; consumers extract messages from queues established to hold their messages. A message is delivered to only one message consumer.

produce Passing a message to the client runtime for delivery to a destination.

producer An object (MessageProducer) created by a session that is used for sending messages to a destination. In the point-to-point delivery model, a producer is a sender (QueueSender); in the publish/subscribe delivery model, a producer is a publisher (TopicPublisher).

publish/subscribe delivery model Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to a topic. The system distributes messages arriving from a topic's multiple publishers to its multiple subscribers.

queue An object created by an administrator to implement the point-to-point delivery model. A queue is always available to hold messages even when the client that consumes its messages is inactive. A queue is used as an intermediary holding place between producers and consumers.

session A single threaded context for sending and receiving messages. This can be a queue session or a topic session.

topic An object created by an administrator to implement the publish/subscribe delivery model. A topic may be viewed as node in a content hierarchy that is responsible for gathering and distributing messages addressed to it. By using a topic as an intermediary, message publishers are kept separate from message subscribers.

transaction An atomic unit of work which must either be completed or entirely rolled back.

user group The group to which the user of a MQ client belongs for purposes of authorizing access to MQ message server resources, such as connections and destinations.

Index

A

access control file
  access rules 214
  format of 213
  location 292, 293, 294
  use for 212
  version 212
access rules 214
acknowledgements
  about 45, 59
  broker 59
  client 60, 88
  delivery, of 60
  transactions, and 60
  wait period for 187
admin connection service 55, 164
administered objects
  about 39, 88
  attributes of 187
  connection factory, *See* connection factory administered objects
  deleting 198
  destination, *See* destination administered objects
  listing 199
  look up name for 191
  object stores, *See* object stores
  provider-independence 89
  querying 200
  queue, *See* queues
  required information 192
  topic, *See* topics
  types 40, 88, 186
  updating 200
  XA connection factory, *See* connection factory administered objects
administration tasks
  development environments 93
  production environments 94
administration tools
  about 96
  Administration Console 96
  command line utilities 97
API documentation 29, 292, 293, 294
application servers, and MQ 42
applications, *See* client applications
attributes
  administered objects 187
  destinations 171
authentication
  about 67
  managing 202
authorization
  about 67
  managing 212
  user groups 68
  See also access control file
auto-create destinations
  about 78
  properties 79

B

benchmarks, performance 227
bottlenecks, performance 230
broker clusters
  adding brokers to 144
  architecture 242
  architecture of 82
  cluster configuration file 85, 141
  configuration change record 84
  configuration properties 85, 141
  connecting brokers 143
  in development-only environments 85
  Master Broker 84, 85
  option to specify 137
  performance impact of 242
  propagation of information in 83
  reasons for using 81, 242
  restarting a broker in 145
  secure inter-broker connections 143
  setting properties 142
  synchronizing clocks 140
broker instances, *See* brokers
broker metrics
  logger properties 74, 249
  metric quantities 256
  metrics messages 73
  reporting interval, logger 138
  using broker log files 250
  using imqcmd 162, 247, 248
  using message-based monitoring 251
broker monitoring service
  about 70
  properties 74
brokers
  about 52
  access control, *See* authorization
  acknowledgements (Ack) 59, 187
  auto-create destination properties 79
  clusters, *See* broker clusters
  components and functions 53
  configuration files, *See* configuration files
  connecting to 156
  connecting together 143
  connection services, *See* connection services
  controlling state of 160
  displaying properties of 159
  HTTP support for 307
  httpjms connection service properties 309
  HTTPS, support for 317
  httpsjms connection service properties 321
  instance configuration properties 130
  instance name 138
  interconnected, *See* broker clusters
  JDBC support, *See* JDBC support
  limit behaviors 61, 242
  listing connection services 164
  logging, *See* logger
  managing 157
  Master Broker 84
  memory management 61, 170, 242
  message capacity 62, 134, 160
  message flow control, *See* message flow control
  message routing, *See* message router
  metrics, *See* broker metrics
  monitoring, *See* broker monitoring service
  multi-broker clusters, *See* broker clusters
  pausing 158, 161
  persistence manager, *See* persistence manager
  properties 160
  querying 159
  recovery from failure 63
  restarting 63, 158, 161
  resuming 158, 161
  security manager, *See* security manager
  shutting down 161
  starting 135
  starting an SSL-based service 221
  updating properties of 160
  Windows service, running as 331
built-in persistence 64

C

certificates 219, 317
client
  applications, *See* client applications
  identifiers (ClientID) 44
  programming model 38
  runtime, *See* client runtime

client applications
example 29, 292, 293, 294
factors impacting performance 230
provider-independence 43
system properties, and 92
client runtime
about 86
configuration of 243
message flow tuning 287
Cloudscape 295
cluster configuration file 85
cluster connection service
port number for 141
setting up, secure 143, 218
clusters, *See* broker clusters
command files 193
command line syntax 99
command line utilities
about 97
basic syntax 99
imqbrokerd, *See,* imqbrokerd command
imqcmd, *See,* imqcmd command
imqdbmgr *See,* imqdbmgr command
imqkeytool, *See,* imqkeytool command
imqobjmgr, *See,* imqobjmgr command
imqsvcadmin, *See,* imqsvcadmin command
imqusermgr, *See,* imqusermgr command
options common to 100
command options 100
components
EJB 40
MDB 41
configuration change record 84
configuration files
default 128
editing 130
installation 128
instance 128, 142, 160, 291, 292, 293
location 291, 292, 293
template location 291, 292, 293
templates 291, 292, 293
connecting, to brokers 156
connection factory administered objects
about 90
adding 195
attributes 90, 187
ClientID, and 44
introduced 39
JNDI lookup 40
overrides 92
connection service metrics
metric quantities 258
using imqcmd metrics 166, 247
using imqcmd query 249
connection services
about 54
access control for 69
activated at startup 57
admin 55, 164
cluster 143, 218
commands affecting 163
connection type 54
displaying properties of 165
HTTP, *See* HTTP connections
httpjms 54, 164
HTTPS, *See* HTTPS connections
httpsjms 55, 164
jms 54, 164
metrics data, *See* connection service metrics
pausing 163, 166
port mapper, *See* port mapper
properties 57, 165
querying 163, 167
resuming 163, 166
service type 54
ssladmin, *See* ssladmin connection service
SSL-based 220
ssljms, *See* ssljms connection service
static ports for 57
thread allocation 165
thread pool manager 56
updating 163, 165, 167
connections
commands affecting 167
introduced 39
listing 167
performance impact of 239
querying 167

consumers 39
containers
  EJB 41
  MDB 41
control messages 59

D

data store
  about 63
  flat-file 64
  JDBC-accessible 65
  location 291, 292, 293
  performance impact of 243
  resetting 139
data, MQ, location of 291
delivery modes
  non-persistent 45
  performance impact of 232
  persistent 45
delivery, reliable 45
destination administered objects
  about 91
  attributes 189
  introduced 39
destination metrics
  metric quantities 260
  using imqcmd metrics 169, 245, 247
  using imqcmd query 249
  using message-based monitoring 251
destinations
  access control 216
  attribute values 173
  attributes of 171
  auto-created 78, 217
  batching messages for delivery 80, 172
  compacting file-based data store 168
  creating 170
  destroying 168, 170, 176
  displaying attribute values 173
  getting information about 169
  information about 173
  introduced 52
  limit behaviors 61, 170, 171
  listing 169, 173
  managing 168
  metrics, *See* destination metrics
  pausing 169, 175
  physical 76
  purging messages from 169, 176
  queue, *See* queues
  restricted scope in cluster 80, 172
  resuming 169, 175
  temporary 81, 173
  topic, *See* topics
  types 76, 169
  updating attributes 170
directory variables
  IMQ_HOME 26
  IMQ_JAVAHOME 27
  IMQ_VARHOME 27
distributed transactions
  about 46
  XA resource manager 46, 180
  See also XA connection factories
domains 43
durable subscriptions
  performance impact of 234
durable subscribers, *See* durable subscriptions
durable subscriptions
  about 43
  ClientID, and 44
  destroying 179, 180
  id 154
  listing 179
  managing 179
  purging messages for 179

E

editions, product
  about 32
  enterprise 33
  platform 33
encryption
  about 68
  Key Tool, and 69
  SSL-based services, and 218

enterprise edition 33
environment variables, *See* directory variables
example applications 29, 292, 293, 294

F

failover 81
file descriptor limits 338
firewalls 56, 305
flow control, *See* message flow control

H

hardware, performance impact of 239
HTTP
  connection service, *See* httpjms connection service
  proxy 305
  support architecture 305
  transport driver 305
HTTP connections
  multiple brokers, for 311
  request interval 309
  support for 305
  tunnel servlet, *See* HTTP tunnel servlet
HTTP tunnel servlet
  about 306
  deploying 307
httpjms connection service
  about 54, 164
  configuring 308
  setting up 307
HTTPS
  connection service, *See* httpsjms connection service
  support architecture 305
HTTPS connections
  multiple brokers, for 323
  request interval 321
  support for 305
  tunnel servlet, *See* HTTPS tunnel servlet
HTTPS tunnel servlet
  about 306
HTTPS tunnel servletLdeploying 318
httpsjms connection service
  about 55, 164
  configuring 320
  setting up 317

I

imq.accesscontrol.enabled property 69, 130
imq.accesscontrol.file.filename property 70, 130
imq.authentication.basic.user_repository property 69, 130
imq.authentication.client.response.timeout property 69, 130
imq.authentication.type property 69, 130
imq.autocreate.destination.isLocalOnly property 80, 130
imq.autocreate.destination.limitBehavior property 79, 130
imq.autocreate.destination.maxBytesPerMsg property 79, 130
imq.autocreate.destination.maxCount property 79, 130
imq.autocreate.destination.maxNumMsgs property 79
imq.autocreate.destination.maxNumProducers property 79, 130
imq.autocreate.destination.maxTotalMsgBytes property 79, 130
imq.autocreate.queue property 79, 130, 160
imq.autocreate.queue.consumerFlowLimit property 80, 131
imq.autocreate.queue.localDeliveryPreferred property 80, 131
imq.autocreate.queue.maxNumActiveConsumers property 80, 131, 160
imq.autocreate.queue.maxNumBackupConsumers property 80, 131, 160
imq.autocreate.topic property 79, 131, 160
imq.cluster.brokerlist property 141

imq.cluster.masterbroker property 141
imq.cluster.port property 141
imq.cluster.transport property 141
imq.cluster.url property 141, 160
imq.hostname property 57, 131
imq.httpjms.http.connectionTimeout property 310
imq.httpjms.http.pullPeriod property 309
imq.httpjms.http.servletHost property 309
imq.httpjms.http.servletPort property 309
imq.httpsjms.https.connectionTimeout property 321
imq.httpsjms.https.pullPeriod property 321
imq.httpsjms.https.servletHost property 321
imq.httpsjms.https.servletPort property 321
imq.keystore.file.dirpath property 220
imq.keystore.file.name property 220
imq.keystore.password property 220, 224
imq.log.console.output property 75, 131
imq.log.console.stream property 75, 131
imq.log.file.dirpath property 74, 131
imq.log.file.filename property 74
imq.log.file.name property 131
imq.log.file.output property 74, 131
imq.log.file.rolloverbytes property 74, 131, 160
imq.log.file.rolloversecs property 74, 131, 160
imq.log.level property 74, 131, 160
imq.log.syslog.facility property 75, 131
imq.log.syslog.identity property 75, 131
imq.log.syslog.logconsole property 75, 132
imq.log.syslog.logpid property 75, 132
imq.log.syslog.output property 75, 132
imq.log.timezone property 75, 132
imq.message.expiration.interval property 62, 132
imq.message.max_size property 62, 132, 160
imq.metrics.enabled property 74, 132
imq.metrics.interval property 74, 132
imq.metrics.topic.enabled property 75, 132
imq.metrics.topic.interval property 76, 132
imq.metrics.topic.persist property 76, 132
imq.metrics.topic.timetolive property 76, 132
imq.passfile.dirpath property 70, 132
imq.passfile.enabled property 70, 132
imq.passfile.name property 70, 132
imq.persist.file.destination.message.filepool.limit property 66, 132
imq.persist.file.message.cleanup property 66, 132
imq.persist.file.message.filepool.cleanratio property 66, 132
imq.persist.file.message.max_record_size property 66, 132
imq.persist.file.message.vrfile.max_record_size property 64
imq.persist.file.sync.enabled property 66, 132
imq.persist.jdbc.brokerid property 298
imq.persist.jdbc.closedburl property 298
imq.persist.jdbc.createdburl property 298
imq.persist.jdbc.driver property 298
imq.persist.jdbc.needpassword property 299
imq.persist.jdbc.opendburl property 298
imq.persist.jdbc.password property 224, 299
imq.persist.jdbc.table.IMQCCREC35 property 299
imq.persist.jdbc.table.IMQDEST35 property 299
imq.persist.jdbc.table.IMQINT35 property 299
imq.persist.jdbc.table.IMQLIST35 property 300
imq.persist.jdbc.table.IMQMSG35 property 300
imq.persist.jdbc.table.IMQPROPS35 property 300
imq.persist.jdbc.table.IMQSV35 property 299
imq.persist.jdbc.table.IMQTACK35 property 300
imq.persist.jdbc.table.IMQTXN35 property 300
imq.persist.jdbc.user property 298
imq.persist.store property 66, 133, 298
imq.ping.interval property 57, 133
imq.portmapper.backlog property 57, 133
imq.portmapper.hostname property 57, 133
imq.portmapper.port property 57, 133, 160
imq.protocol protocol_type inbufsz 282
imq.protocol protocol_type nodelay 282
imq.protocol protocol_type outbufsz 282
imq.*resource_state*.count property 63, 133
imq.*resource_state*.threshold property 63, 133
imq.service.activelist property 57, 133
imq.*service_name*.accesscontrol.enabled property 70, 133

imq.*service_name*.accesscontrol.file.filename property 70, 133
imq.*service_name*.authentication.type property 69, 133
imq.*service_name*.max_threads property 58, 133
imq.*service_name*.min_threads property 58, 133
imq.*service_name.protocol_type*.hostname property 58, 134, 141
imq.*service_name.protocol_type*.port property 57, 134
imq.*service_name*.threadpool_model property 58, 134
imq.shared.connectionMonitor_limit property 58, 134
imq.system.max_count property 62, 134, 160
imq.system.max_size property 62, 134, 160
imq.transaction.autorollback property 63, 134, 182
imq.user_repository.ldap.base property 210
imq.user_repository.ldap.gidattr property 211
imq.user_repository.ldap.grpbase property 211
imq.user_repository.ldap.grpfiltler property 211
imq.user_repository.ldap.grpsearch property 211
imq.user_repository.ldap.memattr property 211
imq.user_repository.ldap.password property 210, 224
imq.user_repository.ldap.principal property 210
imq.user_repository.ldap.server property 210
imq.user_repository.ldap.ssl.enabled property 211
imq.user_repository.ldap.timeout property 211
imq.user_repository.ldap.uidattr property 210
imq.user_repository.ldap.usrfilter property 210
IMQ_HOME directory variable 26
IMQ_JAVAHOME directory variable 27
IMQ_VARHOME directory variable 27
imqAckOnAcknowledge attribute 187
imqAckOnProduce attribute 187
imqAckTimeout attribute 187
imqAddressList attribute 187
imqAddressListBehavior attribute 187
imqAddressListIterations attribute 187
imqbrokerd command
  about 98
  command syntax 135
  options 136
  using 135
imqBrokerHostName attribute (MQ 3.0) 187
imqBrokerHostPort attribute (MQ 3.0) 187
imqBrokerServicetPort attribute (MQ 3.0) 187
imqcmd command
  about 98
  command syntax 152
  connecting to a broker 156
  destination management 168
  metrics monitoring 244
  options 154
  secure connection to broker 155, 223
  subcommands 152
  transaction management 180
  used for 152
imqConfiguredClientID attribute 187
imqConnectioinFlowLimit attribute 187
imqConnectionFlowCount attribute 187
imqConnectionFlowLimitEnabled attribute 187
imqConnectionType attribute (MQ 3.0) 187
imqConnectionURL attribute (MQ 3.0) 188
imqConsumerFlowLimit attribute 188
imqConsumerFlowThreshold attribute 188
imqdbmgr command
  about 98
  command syntax 302
  options 303
  subcommands 302
imqDefaultPassword attribute 188
imqDefaultUsername attribute 188
imqDestinationDescription attribute 91, 189
imqDestinationName attribute 91, 189
imqDisableSetClientID attribute 188
imqFlowControlLimit attribute 188
imqJMSDeliveryMode attribute 188
imqJMSExpiration attribute 188
imqJMSPriority attribute 188

imqkeytool command
about 98
command syntax 219, 318
using 219, 318
imqLoadMaxToServerSession attribute 188
imqobjmgr command
about 98
command syntax 189
options 190
subcommands 190
used for 189
imqOverrideJMSDeliveryMode attribute 188
imqOverrideJMSExpiration attribute 188
imqOverrideJMSHeadersToTemporaryDestinations attribute 188
imqOverrideJMSPriority attribute 188
imqQueueBrowserMax MessagesPerRetrieve attribute 188
imqQueueBrowserRetrieveTimeout attribute 188
imqReconnectAttempts attribute 188
imqReconnectEnabled attribute 188
imqReconnectInterval attribute 188
imqSetJMSXAppID attribute 188
imqSetJMSXConsumerTXID attribute 188
imqSetJMSXProducerTXID attribute 188
imqSetJMSXRcvTimestamp attribute 188
imqSetJMSXUserID attribute 188
imqSSLIsHostTrusted attribute (MQ 3.0) 188
imqsvcadmin command
about 98
command syntax 332
options 333
subcommands 332
used for 332
imqusermgr command
about 98
command syntax 204
options 204
passwords 207
subcommands 204
use for 203
user names 207
instance configuration files, *See* configuration files

J

J2EE applications
EJB specification 40
JMS, and 40
message-driven beans, *See* message-driven beans
Java Virtual Machine, *See* JVM
JDBC support
about 65
driver 295, 298
setting up 295
JDK
option to specify path to 154, 191, 333
specify path to 137
JMS
message structure 37
programming model 38
specification 29, 31, 37
jms connection service 54, 164
JNDI
administered objects, and 40, 43
initial context 184, 186
location (provider URL) 184, 186
lookup 88, 92, 116, 192
lookup name 192, 196
message-driven beans, and 42
MQ support of 32
object store 98, 184
object store attributes 184, 192
JVM
metrics, *See* JVM metrics
performance impact of 239
tuning for performance 281
JVM metrics
metric quantities 255
using broker log files 250
using imqcmd metrics 246
using message-based monitoring 251

K

key pairs
generating 219
regenerating 220

Key Tool 69
keystore
  file 219, 220, 318
  properties 220

L

LDAP server
  object store attributes 184
  user-repository access 209
licenses
  for MQ editions 32
  starting with trial Enterprise Edition license 136
  startup option 138
limit behaviors
  broker 61
  destinations 61, 170, 171
listeners 39, 41
load-balanced queue delivery
  about 76
  attributes 80
  tuning for performance 286
log files
  default location 72, 291, 292, 293
  rollover criteria 74
logger
  about 71
  as broker component 54
  categories 71
  changing configuration 149
  levels 71, 74, 138
  message format 148
  metrics information 74
  output channels 71, 149, 249
  redirecting log messages 150
  rollover criteria 150
  writing to console 75, 139
logging, *See* logger

M

managing
  brokers 157
  destinations 168
Master Broker 84, 85
MDB, *See* message-driven beans
memory management
  for broker 61
  tuning for performance 285
  using destination attributes 170
message consumers, *See* consumers
message delivery models 37, 43
message flow control
  broker 61, 170
  limits 287
  metering 287
  performance impact of 243
  tuning for performance 287
message listeners, *See* listeners
message producers, *See* producers
message router
  about 58
  as broker component 53
  properties 62
message server
  about 52
  architecture 242
  multi-broker, *See* broker clusters 81
message service
  about 36
  factors impacting performance 238
message-driven beans
  about 41
  application server support 42
  deployment descriptor 42
  MDB container 41
messages
  acknowledgements 60
  body type, and performance 237
  broker limits on 62, 134, 160
  consumption of 87
  control 59
  delivery flow, and bottlenecks 229
  delivery models 37, 43
  delivery modes, *See* delivery modes

messages (continued)
destination limits on 171
filtering, *See* selectors
flow control, *See* message flow control
introduced 37
latency 226
listeners for 39, 87
load-balanced queue delivery 76
metrics 72
metrics messaes, *See* metrics messages
ordering 48
persistence of 61, 63
persistent 45
point-to-point delivery 43
prioritizing 48
production of 87
publish/subscribe delivery 44
purging from a destination 169
reclamation of expired 62
redelivery 60
reliable delivery of 45
routing and delivery 59
size, and performance 236
SOAP 32
structure 37
throughput performance 226
messaging system
architecture 36
message service 36
MQ architecture 52
metrics
about 71
data, *See* metrics data
messages, *See* metrics messages
monitoring tools *See* metrics monitoring tools
topic destinations 73, 251
metrics data
broker, *See* broker metrics
connection service, *See* connection service metrics
descriptive lising of 255
destination, *See* destination metrics
using broker log files 249
using imqcmd metrics 246
using message-based monitoring API 251
metrics messages
about 72, 251
contents of 73
type 73, 251
metrics monitoring tools
compared 253
message-based monitoring API 250
MQ Command Utility (imqcmd) 244
MQ log files 249
monitoring, *See* performance monitoring

O

object stores
about 184
file-system store 185
file-system store attributes 186
LDAP server 184
LDAP server attributes 184
locations 291, 292, 294
operating system
performance impact of 239
tuning Solaris performance 280
Oracle 295

P

passfile
broker configuration properties 70
command line option 138
location 224, 292, 293, 294
using 224
password file, *See* passfile
passwords
encoding of 69
JDBC 224
LDAP 224
naming conventions 207
passfile, *See* passfile
SSL keystore 138, 220, 224
passwords, default 188

pausing
brokers 158, 160, 161
connection services 163, 166
destinations 169, 175
performance
about 225
baseline patterns 228
benchmarks 227
bottlenecks 230
factors impacting, *See* performance impact factors
indicators 226
measures of 226
monitoring, *See* performance monitoring
optimizing, See performance tuning
reliability trade-offs 47, 231
troubleshooting 262
tuning, *See* performance tuning
performance impact factors
acknowledgement mode 234
broker limit behaviors 242
connections 239
data store 243
delivery mode 232
durable subscriptions 234
hardware 239
JVM 239
message body type 237
message flow control 243
message server architecture 242
message size 236
operating system 239
selectors 236
transactions 233
transport protocols 240
performance monitoring
metrics data, *See* metrics data
tools, *See* metrics monitoring tools 244
performance tuning
boker adjustments 285
client runtime adjustments 286
process overview 225
system adjustments 280
permissions
access control properties file 67, 213
admin service 68
computing 214
data store 65
embedded database 297
keystore 318
MQ operations 67
passfile 224
user repository 204
persistence
built-in 64
data store *See* data store
delivery modes, *See* delivery modes
JDBC, *See* JDBC persistence
persistence manager, *See* persistence manager
plugged-in, *See* plugged-in persistence
persistence manager
about 63
as broker component 54
data store, *See* data store
JDBC data store 297
plugged-in persistence, and 295
properties 66
persistent messages 45
platform edition 33
plugged-in persistence
about 65
setting up 295
tuning for performance 285
PointBase 295
point-to-point delivery 43
port mapper
about 55
port assignment for 57, 138
portability, *See* provider-independence
ports, dynamic allocation of 56
producers
about 39
destination limits on 79, 171
programming domains 43
properties
administered objects, *See* administered objects, attributes of
auto-create 79
broker instance configuration 130
broker monitoring service 74
broker, updating 160
cluster configuration 141

properties (continued)
connection service 57
destinations, *See* destinations, attributes of
httpjms connection service 309
httpsjms connection service 321
JDBC-related 298
keystore 220
logger 74
memory management 62, 170
message router 62
persistence 66
security 69
protocol types
HTTP 54, 164
TCP 54, 164
TLS 54, 164
protocols, *See* transport protocols
provider-independence
about 43
administered objects 89
publish/subscribe delivery 44
purging, messages from destinations 176

Q

querying
brokers 159
connection services 163, 165, 167
queue destinations, *See* queues
queue load-balanced delivery
attributes 171
queues 76
adding administered objects for 197
attributes of 171
auto-created 79, 130
load-balanced delivery, *See* load-balanced queue delivery

R

redeliver flag 60
reliable delivery 45
performance trade-offs 47, 231
resource adapter 42
restarting brokers 158, 161
resuming
brokers 158, 160, 161
connection services 163, 166
destinations 175
routing, *See* message router

S

Secure Socket Layer standard, *See* SSL
security
authentication, *See* authentication
authorization, *See* authorization
encryption, *See* encryption
manager, *See* security manager
object store, for 184
security manager
about 66
as broker component 54
properties 69
selectors
about 48
as message properties 38
performance impact of 236
self-signed certificates 219, 317
service types
ADMIN 54
NORMAL 54
servlet, tunnel, *See* HTTP/HTTPS tunnel servlet
sessions
acknowledgement options for 45
introduced 39
transacted 45
shutting down brokers 158, 161

Simple Object Access Protocol *See* SOAP
SOAP 32
SSL
  about 68
  connection services, *See* SSL-based connection services
  encryption, and 218
  over HTTP 224
  over TCP/IP 218
ssladmin connection service
  about 55, 164
  setting up 218
SSL-based connection services
  setting up 201, 218
  starting up 221
ssljms connection service
  about 54, 164
  setting up 218
starting
  brokers 135
  brokers in a cluster 145
  SSL-based connection services 221
subscriptions
  about 43
  durable, *See* durable subscriptions
syslog 72, 150
system properties, setting 92

T

TCP 54, 164
temporary destinations 81, 173
thread pool manager
  about 56
  dedicated threads 56
  shared threads 56
TLS 54, 164
tools, administration, *See* administration tools
topic destinations, *See* topics
topics
  about 44
  adding administered objects for 196
  as physical destinations 78
  attributes of 171
  auto-created 79, 130
transactions
  about 45
  acknowledgements, and 60
  committing 180
  distributed, *See* distributed transactions
  information about 180
  managing 180
  performance impact of 233
  rolling back 180
transport protocols
  performance impact of 240
  protocol types, See protocol types
  relative speeds 240
  tuning for performance 281
troubleshooting 262
tunnel servlet *See* HTTP/HTTPS tunnel servlet

U

updating
  brokers 160
  connection services 163, 165, 167
user groups
  about 67
  default 68
  deleting assignment 206
  predefined 205
user names 188
  default 203
  format 207
user repository
  about 67
  flat-file 202
  LDAP server 209
  location 292, 293, 294

user repository (continued)
managing 207
platform dependence 204
populating 207
property 69
user groups 206
user states 206

W

Windows service, broker running as 331

X

XA connection factories
about 47
See also connection factory administered objects
XA resource manager, *See* distributed transactions