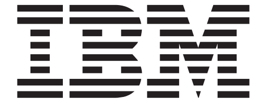
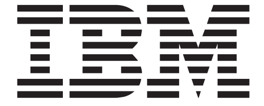
WebSphere MQ Everyplace



MQe Application Programming

Version 2 Release 0

WebSphere MQ Everyplace



MQe Application Programming

Version 2 Release 0

Note

Before using this information and the product it supports, read the information in the Notices appendix.

First Edition (July 2004)

This edition applies to IBM WebSphere® MQ Everyplace Version 2.0.1 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2000, 2004. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this topic collection ix

Developing a basic application 1
Introduction to the MQe development kit. 1
Setting up your development environment 1
Java development. 1
J2ME environment 3
C development 3
Using embedded Visual C++ 5
Threading 6
Calling conventions 6
Handles and items 6
MQe memory functions. 7
MQeString 7
Walkthrough: creating a basic application 10
1. Create a queue manager (QM1) 10
2. Start the queue manager (QM1) 11
3. Create a local queue (Q1) 12
4. Create a connection definition 12
5. Create a remote queue definition 12
6. Create a listener (L1) 13
7. Start listener (L1). 13
8. Create a second queue manager (QM2) . . . 13
9. Start QM2 14
10. Create a local queue (on QM2) called Q2 . . 14
11. Create a connection definition (on QM2) . . 15
12. Create a remote queue definition (on QM2) 15
13. Create a listener (on QM2) called L2 15
14. Start the listener L2 (on QM2) 16
15. Send (PUT) a message from QM1 to QM2 . . 16
16. Receive (GET) the message on QM2 16
17. Displaying details of MQe objects. 17
An example MQe application (HelloWorld). . . . 17
Java ″HelloWorld″ 17
Designing the Java application 17
Developing the Java application 18
Overview of examples.helloworld.run . . 18
Start the queue manager 18
Create a message and put to a local queue 19
Get message from a local queue 20
Stopping and deleting the queue manager 20
Running the Java application 20
C ″HelloWorld″ 21
Designing the C application 21
Developing the C application 21
Overview of HelloWorld_Runtime.c . . . 21
Start the queue manager 22
Create a message 23
Put message to a local queue 23
Get message from a local queue 23
Shutdown 24
Compiling. 24
Deploying the C application. 24
Running the C application 25
Using the MQe development and administration tools. 25
Using WebSphere Studio Device Developer (WSDD) 26
Developing applications for Palm 26
Developing applications for PocketPC . . . 28
Debugging applications 29
Runnable classes. 30
MIDlets. 30
Cleaning up after applications 31
Constraints of SmartLinker 31
Further information 32

Designing your real application 33
Messaging 33
What are MQe messages? 33
Message properties 34
Symbolic names 34
Examples 35
Message filters 36
Message expiry 37
Checking for expired messages 37
Assurance of expiry 38
MQeFields. 38
Storage and retrieval of values in MQeFields 39
Embedding MQeFields items 40
Queues 40
What are MQe queues? 40
Queue names. 41
Queue properties 41
Queue types 42
Local queue 42
Remote queue 43
Store-and-forward queue 43
Dead-letter queue 44
Administration queue 44
Home-server queue. 45
MQ bridge queue 45
Queue persistent storage 45
MQe connection definitions 45
Using queue aliases 48
Examples of queue aliasing 48
Merging applications 48
Upgrading applications 48
Using different transfer modes to a single queue 49
Queue manager operations 49
What is an MQe queue manager 49
The queue manager life-cycle 50
Creating queue managers. 50
Queue manager names 51
Creating a queue manager - step by step . . 51
Create and activate an instance of MQeQueueManagerConfigure 51
Set queue manager properties 52

© Copyright IBM Corp. 2000, 2004

Create definitions for the default queues. . 53
Close the MQeQueueManagerConfigure instance 54
Persistent configuration data 54
Creating simple queue managers 55
Creating a simple queue manager in Java 55
Creating a simple queue manager in C . . 56
Starting queue managers 57
Starting queue managers in Java 57
Starting a simple queue manager in Java 57
Starting queue managers in C 58
Starting a simple queue manager in C . . 58
Queue manager parameters 59
Registry parameters for a queue manager . . 61
Registry type 61
Client queue managers 61
Example - starting a client queue manager 61
Example - MQePrivateClient 63
Server queue managers 63
Example - MQeServer 63
Example - MQePrivateServer 64
Environment relationship 65
Java code 65
C code 65
Stopping queue managers 65
Stopping a queue manager in Java. 65
closeQuiesce 65
closeImmediate 66
Stopping a queue manager in C 66
Deleting queue managers. 66
Java 66
1. Delete any definitions 67
2. Create and activate an instance of MQeQueueManagerConfigure 67
3. Delete the standard queue and queue manager definitions 67
4. Close the MQeQueueManagerConfigure instance 68
C 68
Messaging lifecycle 69
Message states 69
Message events 70
Message index fields 71
Messaging operations 71
Put 72
Get 72
Delete 72
Browse 73
confirmPut 74
confirmGet 74
Listen 74
Wait 74
Queue ordering 75
Reading messages on a queue 75
Java 75
C 75
Browse and Lock 75
Example - Java 76
Example - C 76
Message listeners 77
Message polling 78
Trigger transmission 78
Trigger transmission rules 79
Servlet 79
Example - configuring a connection on a servlet 79
Example - configuring a connection on a servlet using aliases 79
Differences between server and servlet startup 80
Example - starting a servlet 80
Example - handling incoming requests . . . 81
Running multiple servlets on a web server . . 82
Message delivery 82
Asynchronous message delivery 82
Synchronous message delivery 83
Assured and non-assured message delivery . . 83
Assured message delivery 83
Non-assured message delivery 83
Synchronous assured message delivery 84
Put message - assured put 84
Example (Java) - assured put 85
Example (C) - assured put 85
Exception handling - put message 86
Get message - assured get 88
Example (Java) - assured get. 89
Example (C) - assured get 90
Undo command 91
Network topologies and message resolution . . . 92
Overview 92
Introduction 93
Local queue resolution. 94
Local queue alias 94
Queue manager alias 96
Remote queue resolution 97
Aliases on remote queues 100
Parallel routes 102
Chaining remote queue references 104
Pushing store and forward queues 104
S&F queues and remote queue references . . 106
Chaining S&F queues 107
Home server queues 108
Via connections. 110
Rerouting with queue manager aliases 113
MQe-MQ bridge message resolution. 117
Pulling messages from MQ 118
Single pull route 118
Multiple pull route 120
Pushing messages to MQ 121
Connecting a client to MQ via a bridge. . . 123
Pushing messages to MQ with a via connection 127
Security considerations 130
Resolution rules 130
Rule 1: Resolve queue manager aliases . . . 130
Queue resolution 130
'Exact' match 131
Queue Alias Match 131
S&F queue 131
Queue Discovery 131
Failure 132
Push across network 132
Normal 132

Via 132
Home server pulling 132
Using aliases 132
Using queue aliases 132
Merging applications 133
Upgrading applications 133
Using different transfer modes to a single queue 133
Using queue manager aliases 134
Addressing a queue manager with several different names. 134
Different routings from one queue manager to another 135
Aliasing on the sending side 135
Virtual queue manager on the receiving side 136
Using adapters 137
Storage adapters 137
Communications adapters 138
How to write adapters 139
An example communications adapter 141
An example message store adapter 148
The WebSphere Everyplace Suite (WES) communications adapter 152
The WebSphere Everyplace Suite (WES) adapter files 153
Using the WebSphere Everyplace Suite (WES) adapter 154
General operation 154
Using the authentication dialog example 156
Using the application example. 157
Using rules 158
Queue manager rules. 158
Loading and activating queue manager rules 158
Java example queue manager rule . . . 158
C example queue manager rule 159
Using queue manager rules 160
Example put message rule 160
Example get message rule 161
Example remove queue rule 163
Transmission rules. 163
Trigger transmission rule example 163
Transmit rule 164
Transmit rule - Java example 1 164
Transmit rule - C example 1 165
A more complex transmit rule example. . . 165
Transmit rule - Java example 2 165
Transmit rule - C example 2 168
Activating synchronous remote queue definitions 171
Queue rules 172
Using queue rules 172
Queue rules - Java example 1 173
Queue rules - C example 1 173
Queue rules - Java example 2 174
Queue rules - C example 2 175
Java Message Service (JMS). 177
Using JMS with MQe. 177
Obtaining jar files 178
Testing the JMS class path 178
Running other MQe JMS example programs 179
Writing JMS programs 180
The JMS model. 180
Building a connection 181
Using the factory to create a connection 182
Starting the connection 182
Obtaining a session 182
Sending a message 183
Message types 183
Receiving a message 184
Handling errors 184
Exception listener 185
JMS messages 185
Message selectors 185
Restrictions in this version of MQe 187
Using Java Naming and Directory Interface (JNDI) 188
Storing and retrieving objects with JNDI . . 188
Using the sample programs with JNDI . . . 189
Mapping JMS messages to MQe messages. . . 193
Naming MQeMsgObject fields. 193
MQe JMS information 194
JMS header files 194
JMS properties 195
JMS message body 197
MQe JMS classes 197
Security 198
Levels of security 199
Local security 199
Local security usage scenario 200
Examples - Java 202
Examples - C 203
Message level security 206
Message-level security usage scenario . . 206
Examples - using MAttribute for Java . . 208
Examples - using MAttribute for C . . . 209
Examples - using MTrustAttribute for Java 210
Non-repudiation 212
Queue-based security. 213
Security properties 213
Effects of queue attributes 214
Configuring queue-based security . . . 214
Queue manager based security 226
Configuring queue manager security . . 226
Setting up the queue manager. 226
Setting up a private registry 226
Channel level security 230
Channel attribute rules 231
Certificate management 233
Examining certificates 233
Renewing certificates 235
Security services 236
Private registry service 236
Private registries 236
Private registry usage guide 237
Private registry usage scenario 238
Private registry and authenticatable entity 238
Public registry service 240
Public registry usage scenario 240
Secure feature choices 240
Selection criteria 240
Example - public registry 240

Mini-certificate issuance service 241
Renewing mini-certificates 242
Obtaining new credentials (private and public keys) 243
Listing mini-certificates 243
Performance. 244
Errors and error handling 245
Error handling in Java 245
Error handling in C 245
Code structure 245
Exception block 245
Useful macros 246
Java programming samples. 247
Adapters (examples.adapters) 247
Command line administration (examples.administration.commandline) . . . 247
GUI administration (examples.administration.console) 248
Simple administration (examples.administration.simple) 248
Interaction with a queue manager (examples.application) 248
Security (examples.attributes) 249
Adding a small GUI to an application (examples.awt) 250
Managing mini-certificates (examples.certificates) 251
Logging events (examples.eventlog) 251
Creating and deleting queue managers (examples.install) 251
Extending the MQ bridge (examples.mqbridge.awt) 252
Administering objects for an MQ bridge (examples.mqbridge.administration.commandline)253
Testing communication between MQ and MQe (examples.mqbridge.application.GetFromMQ) . 253
MQe interface (examples.mqeexampleapp) . . 253
JNI implementation (examples.nativecode) . . 254
Running a QM as a client, server, or servlet (examples.queuemanager) 254
Rules classes (examples.rules) 255
Trace handling (examples.trace) 255

Deploying your application 257

Packaging and deployment. 257
Java deployment 257
Supplied jar files 257
Optimizing footprint 258
JMS requirements 266
MQe classes for Java requirements 266
Using WSDD smart linker 266
J2ME Midp specifics 267
4690 specifics 268
Packaging 269
Deployment to devices 270
C deployment 271
Supplied DLLs 271
Open Services Gateway initiative (OSGi) 272
MQe example bundle contents 272
Using MQe within OSGi 272
Running the example bundles 273
Server application (MQeServerBundle.jar) 273
Client application (MQeClientBundle.jar) . . 273
Running the example. 274
Providing user-defined rules and dynamic class loading 275

Problem solving 277

Problem determination 277
Common problems 277
Tracing and logging 278
Tracepoints generated from MQe 278
Tracing and logging with Java. 278
Generating trace information (Java) 278
Capturing trace information (Java) 279
Writing your own trace handler (Java) . . . 280
Tracing and logging with C 281
Trace architecture (C). 281
Configuring trace (C). 281
MQe Diagnostic tool 282
Windows diagnostics. 283
Unix diagnostics 283
Other systems diagnostics 284
Information required by IBM support 284

Programming reference 287

JMX Attributes and operations 287
Admin MBean 288
Attributes 288
Operations 289
Queue manager 289
Attributes 289
Operations 290
Operations parameters 292
Remote queue manager 293
Attributes 293
Operations 294
Operations parameters 295
Admin queue 296
Attributes 296
Operations 297
Operations parameters 298
Application queue. 298
Attributes 298
Operations 299
Operations parameters 299
Home Server queue 299
Attributes 299
Operations 300
Asynchronous Proxy queue 300
Attributes 300
Operations 302
Operations parameters 302
Synchronous Proxy queue 302
Attributes 302
Operations 303
Operations parameters 303
Store queue 303
Attributes 304
Operations 305
Operations parameters 305

Forward queue 305
Attributes 306
Operations 307
Operations parameters 307
Communications Listener 307
Attributes 307
Operations 308
MQ/Alias connection 308
Attributes 308
Operations 309
Operations parameters 309
Direct connection 309
Attributes 309
Operations 310
Operations parameters 310
Indirect connection 310
Attributes 311
Operations 311
Operations parameters 311
MQ Bridge queue 311
Attributes 311
Operations 312
Operations parameters 313
MQ Bridge 313
Attributes 313
Operations 314
Operations parameters 314
MQ Queue Manager Proxy 314
Attributes 314
Operations 315
Operations parameters 315
MQ Client Connection 315
Attributes 315
Operations 316
Operations parameters 317
MQ Listener. 317
Attributes 317
Operations 318

Glossary 319

Appendix. Notices 325
Trademarks 326

About this topic collection

This PDF collection has been created from the source files used to create the WebSphere MQ Everyplace Help Center, for when you need a printed copy.

The content of these topics was created for viewing on-screen; you might find that the formatting and presentation of some figures, tables, examples, and so on, is not optimized for the printed page. Text highlighting might also have a different appearance.

In this PDF, links within the topic content itself are included, but are active only if they link to another topic in the same PDF collection (when the link includes a page number). Links to topics outside this topic collection attempt to link to a PDF that is named after the topic identifier (for example, des10030.pdf) and therefore fail; you can identify invalid links like this because they have no associated page number. Use the Help Center to navigate freely between topics.

Please do not provide feedback on this PDF. Refer to the help center, and use the ″Feedback on the documentation″ topic at the end of the table of contents to report any errors or suggestions for improvement.

© Copyright IBM Corp. 2000, 2004

Developing a basic application

This topic contains the information that you need for creating a simple MQe application. It introduces the MQe development toolkit and explains what you need to do to set up your development environment. The walkthrough then gives step-by-step instructions on how to create a simple MQe application, and verify that it is working.

A simple example application called HelloWorld is also described. This simple application demonstrates how to use some of the features of MQe. Finally, the topic introduces some of the tools that you can use to develop and administer MQe applications.

Introduction to the MQe development kit

This topic introduces the MQe Development Kit, which is a development environment for writing messaging and queueing applications based on Java and C. For information on the availability of development kits for environments other than Java and C, see the WebSphere MQ web site at:

`http://www.ibm.com/software/ts/mqseries`

The code portion of the Java development kit comes in two sections:

Base WebSphere MQ Everyplace classes
: A set of Java classes that provide all the necessary function to build messaging and queueing applications.

Examples
: Java source code and classes that demonstrate how to use many features of MQe. Some examples are supplied in "Java programming samples" on page 247.

The code portion of the C development kit also comes in two sections:

Base WebSphere MQ Everyplace functions
: C code that provides all the necessary function to build messaging and queuing applications.

Examples
: C source code that demonstrates how to use the many features of MQe.

Setting up your development environment

This topic provides information on setting up your development environment for Java and C.

Java development

To develop programs in Java using the MQe development kit, you must set up the Java environment as follows:

- Set the *CLASSPATH* so that the Java Development Kit (JDK) can locate the MQe classes.

  Windows

© Copyright IBM Corp. 2000, 2004

In a Windows® environment, using a standard JDK, you can use the following:

```
Set CLASSPATH=<MQeInstallDir>\Java;%CLASSPATH%
```

UNIX®

In a UNIX environment you can use the following:

```
CLASSPATH=<MQeInstallDir>/Java:$CLASSPATH
export CLASSPATH
```

- If you are developing code that uses or extends the MQ–bridge, the MQ Classes for Java must be installed and made available to the JDK.

You can use many different Java development environments and Java runtime environments with MQe. The system configuration for both development and runtime is dependent on the environment used. MQe includes a file that shows how to set up a development environment for different Java development kits. On Windows systems this is a batch file called JavaEnv.bat, for UNIX systems it is a shell script called JavaEnv. To use this file, copy the file and modify the copy to match the environment of the machine that you want to use it on.

A set of batch files and shell scripts that run some of the MQe examples use the environment file described above, and, if you wish to use the example batch files, you must modify the environment file as follows:

- Set the *JDK* environment variable to the base directory of the JDK.
- Set the *JavaCmd* environment variable to the command used to run Java applications.
- If MQ Classes for Java is installed, set the *MQDIR* environment variable to the base directory of the MQ Classes for Java.

Note: Customized versions of JavaEnv.bat or JavaEnv may be overwritten if you reinstall MQe.

When you invoke JavaEnv.bat on Windows you must pass a parameter that determines the type of Java development kit to use.

Possible values are:

Sun - Sun

JB - Borland JBuilder

MS - Microsoft®

IBM - IBM

Note: These parameters are case sensitive and must be entered exactly as shown.

If you do not pass a parameter, the default is IBM.

The JavaEnv shell script on UNIX does not use a corresponding parameter.

On Windows, by default, you must run JavaEnv.bat from the <MQeInstallDir>\java\demo\Windows directory. On UNIX, by default, you must run JavaEnv from the <MQeInstallDir>/Java/demo/UNIX directory. Both files can be modified to allow them to be run from other directories or to use other Java development kits.

J2ME environment

There are two distinct J2ME environments:

Connected Device Configuration (CDC) and Profile
An example is Foundation + Applications in the CDC environment, which can effectively be developed like a normal Java 2 Platform Standard Edition (J2SE) application. The only change required is modifying the bootclasspath option to point to the relevent CDC jar or zip class file.

Note: The 'bootclasspath' option may not be available on all JVM's

Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP)
Applications developed for MIDP can also be compiled using a normal J2SE JVM (again using the bootclasspath to point to the required Midp class library), but they normally have to be run within a Midp Emulator. Therefore, we recommend developing the application using one of the MIDP Toolkits available on the Web. MQe provides a MIDP jar that should be used within this environment. The `MQeMidpBase.jar` is in the <MQeInstallDir>\Java\Jars directory.

C development

To develop programs in C using the MQe Development Kit, you need the following tools:

Microsoft eMbedded Visual C++ (EVC) Version 3.0.
This is included in Microsoft eMbedded Visual Tools 3.0, which is available as a free download from the Microsoft web page:

`http://msdn.microsoft.com/mobile/`

You must use version 3.0 as version 4.0 does not support PocketPC.

An SDK for your chosen platform
Microsoft eMbedded Visual Tools 3.0 includes an SDK for PocketPC 2000. You can also download an SDK for PocketPC 2002 from Microsoft:

`http://msdn.microsoft.com/mobile/`

C Bindings

For the C Bindings codebase see C Bindings Programming Guide - Getting Started.

Native C

For general information see C API Programming Reference, in particular the page *Compilation Information*. However, that page is now slightly out of date and this topic provides an update.

For the native C codebase, support is provided for four platforms:

- PocketPC2000
- PocketPC2002
- PocketPC2003
- Windows 32bit.

For PocketPC, binaries are provided for both the device and the emulator that is available in the Integrated Development Environment Microsoft eMbedded Visual C++. The binaries provided for the devices are compiled for ARM processors.

Binary files

The root of the binary files, as well as the documentation and examples, is the C directory below the directory where you choose to install MQe.

Then in the C directory, the files are located as follows:

PocketPC2000

ARM

DLLs C\PocketPc2000\arm\bin

LIBs C\PocketPc2000\arm\lib

Emulator

DLLs C\PocketPc2000\x86emulator\bin

LIBs C\PocketPc2000\x86emulator\lib

PocketPC2002

ARM

DLLs C\PocketPc2002\arm\bin

LIBs C\PocketPc2002\arm\lib

Emulator

DLLs C\PocketPc2002\x86emulator\bin

LIBs C\PocketPc2002\x86emulator\lib

PocketPC2003

ARM

DLLs C\PocketPc2003\arm\bin

LIBs C\PocketPc2003\arm\lib

Emulator

DLLs C\PocketPc2003\x86emulator\bin

LIBs C\PocketPc2003\x86emulator\lib

Windows 32bit

DLLs C\Win32\Native\bin

LIBs C\Win32\Native\lib

Header files

The header files are common to all the Native platforms, and can be found in the include directory below the installation directory.

MQe_API.h

This is the ″root″ header file. If this is included all relevant header files included for you.

In order to ensure the correct files and definitions are included you must indicate that you are running the Native code base as follows:

```
#define   NATIVE                  // or specify this as an option to the compiler
#include  <published/MQe_API.h>
```

Linking

You need to link against the following two libraries:-

HMQ_nativeAPI.lib
// the API library

HMQ_nativeCnst.lib
// the static constant MQeString library

You need to include both these files. Then an optimizing linker removes links to any functions and constants that you have not used.

The other MQe libraries are statically and dynamically linked with the main API library and are included as required.

Using embedded Visual C++

You can compile applications using the EVC Integrated Development Environment (IDE), or optionally, from the command line. However, you must consider the following:

- Set the appropriate ″Active WCE Configuration″, using the WCE Configuration toolbar. To do this, under **Target Operating System** select either PocketPC or PocketPC 2002. Also, under Target Processor , select one of the following:
  - Win32 (WCE x86em) Debug
  - Win32 (WCE x86em) Release
  - Win32 (WCE ARM) Debug
  - Win32 (WCE ARM) Release

  Note: Some of the Target Processor or **Target Operating System** options may not be available, depending on which SDKs you have installed.
- Include the header files for the native C codebase. These are shared between the two versions of PocketPC and by the C Bindings. The header file location is in the installation directory under `include`. If you include the root header file, MQe_API.h, you include all the functions that you may require. As header files are shared, you need to define which version of the codebase you are using, as shown in the following example:

  ```
  #define NATIVE
  #define MQE_PLATFORM  PLATFORM_WINCE

  /*Alternatively, we recommened that you add this to the Preprocessor Definintions
  in the Project Settings Dialog.  Add the following to the start
  of the list*/
  NATIVE,MQE_PLATFORM=PLATFORM_WINCE

  #include <published\MQe_API.h>
  ```
- Include an entry for the top level MQe include directory in ″Additional include directories″. This varies according to where you install the product.
- Insert the following .lib file names in the ″Project Settings″ dialog, under **Link —> Input** :
  - HMQ_nativeAPI.lib
  - HMQ_nativeCnst.lib

Note: There are variations of these files for each supported release, for example one for PocketPC 2000 ARM, one for PocketPC 2000 x86em, and so on. To ensure that you use the correct verion, qualify the filename fully for each target build.

It is recommended that you develop applications using the PocketPC or PocketPC2002 emulator as this typically provides a faster compilation and debug environment. However, current emulators are API emulators, meaning that they do not emulate ARM hardware. They emulate PocketPC API calls, but the code is still x86, that is running in an x86 virtual machine in the PocketPC 2002 emulator case. Therefore, we recommend that you regularly test the application on the real target device, as many problems such as byte-alignment only becomes apparent on the real device.

Note: MQe emulator binaries are provided only for development purposes and are not suitable for deployment into a production environment.

Threading

The native codebase is designed to be re-entrant. The actual codebase does not use threads, but this does not preclude the use of multiple threads in the application. For example, you can create an application thread to repeatedly call `mqeQueueManager_triggerTransmission()`. If you want to use multiple threads, you do not need to call any specific APIs.

Although it is not a requirement. we recommend that you have an exception block per thread. If you use one exception block shared across threads, an exception block for a thread that fails can be overwritten by the exception block for a thread that succeeds.

Note: You must call `mqeSession_initialize` or `mqeSessuion_terminate`once only, before any threads use an MQe API call. To ensure this, call it in the main thread before any application threads are created. For example, **do not** use the following:

```
mqeSession_initialize();
mqeSession_initialize();
mqeSession_terminate();
mqeSession_terminate();
```

Calling conventions

The calling convention for all of the APIs has been explicitly set at `_cdecl`. However, you can use a different default calling convention in your application.

Handles and items

An application needs a mechanism for accessing MQe items such as the queue manager, fields, strings, and so on. Handles use MQe items. The handle points to an area of memory used to store the specific information for that instance of the item. Type information is held for each item. Therefore, you must take care to initialize the handle correctly.

To use a handle, you must initialize it. You can do this by calling the new function of the associated item to be used. For example, to create an MQeString, you must first call the `mqeString_new()` function and pass a pointer to `MQeStringHndl` to that function. The `mqeString_new()` function allocates memory for the internal structure

and sets the required default values by MQeString. Once completed successfully, the function returns the handle, which can now be used in subsequent calls to MQeString functions.

Once an item has been finished with, it is important to call the `free()` function of the item with which the handle is associated. The `free()` functions release all the systems resources used by that item. Setting the handle to `NULL` introduces a memory leak to the application and the system may run out of resources. To avoid this, set the handle to `NULL` after it has been freed.

Note: We recommend that you do not attempt to free a handle more than once, as this can cause unpredictable results.

You must use handles only with their associated items. You must also initialize and free them in the correct manner. The only instances where the application is not responsible for initializing the handle is when a pointer to a handle is passed as an input parameter to an MQe API. In such instances, a fully initialized handle is returned to the application without the user having to invoke the relevant new() function. An example of this is `mqeQueueManager_BrowseMessages()`, which has a pointer to an `MQeVectorHndl` as an input parameter. However, in instances like this, the application is still responsible for freeing the handle.

MQe memory functions

MQe provides the following functions for memory management:
- `mqeMemory_allocate`
- `mqeMemory_free`
- `mqeMemory_reallocate`

These functions use the same memory management routines that are used within the MQe codebase. These are available for use by application programs. An application can generally use its own choice of memory management. However, some API calls, for example `mqeAdministrator_QueueManager_inquire`, need to return blocks of memory containing information. In this case, the memory must be freed using the `mqeMemory_free function`.

An additional advantage of using the mqeMemory functions is that their use gets traced along with mqe processing. However, never mix the memory allocation calls. For example, do not free memory allocation with `mqeMemory_allocate` with the C runtime `free()` call, as the application can become unstable.

MQeString

The MQeString class contains user defined and system strings. It is an abstraction of character strings used throughout the C API where a string is required. MQeString allows you to create a string in a number of formats, such as arrays containing Unicode code points, with each code point stored in a 1, 2, or 4 byte memory space, and UTF-8. The current implementation of MQeString supports external formats only.

Note: Although they are passed using an MQeString, some API calls require the actual string to lie within the valid ASCII range.

Constant Strings

A number of constant strings are provided. These are defined in the following header files:

- MQe_Admin_Constants.h
- MQe_Adapter_Constants.h
- MQe_Attribute_Constants.h
- MQe_Connection_Constants.h
- MQe_MQe_Constants.h
- MQe_MQeMessage_Constants.h
- MQe_Queue_Constants.h
- MQe_Registry_Constants.h

Constructor

```
MQERETURN osaMQeString_new(MQeExceptBlock* pExceptBlock,
                     MQEVOID*       pInputBuffer,
                     MQETYPEOFSTRING type,
                     MQeStringHndl * phNewString
                          );
```

This function creates a new MQeString object from a buffer containing character data. The data can be in a number of supported formats including, null terminated single byte character arrays (i.e. normal C char* strings), null terminated double-byte Unicode character arrays, null terminated quad-byte Unicode character arrays, and null terminated UTF-8 arrays. The type parameter tells the function what format the input buffer is in.

Destructor

```
MQERETURN osaMQeString_delete(MQeExceptBlock* pExceptBlock,
                              MQeString_*     pString
                             );
```

This function destroys an MQeString object that was created using osaMQeString_new, or MQeString_duplicate, or MQeString_getMQeSubstring

Getter

```
MQERETURN osaMQeString_get(MQeExceptBlock*      pExceptBlock,
                     MQEVOID*            pOutputBuffer,
                     MQEINT32*           pBufferLength,
                     MQETYPEOFSTRING     requiredType,
                     MQECONST MQeStringHndl hString
                     );
```

This function populates a character buffer with the contents of an MQeString performing conversion wherever necessary. Only simple conversions are carried out. No codepage conversion is attempted. For example, if an SBCS string has been put into the string, then trying to get the data out as DBCS (Unicode) data works correctly. If the data was put in as DBCS however, and you try to get the data out as SBCS, this only works if the data does not have any values that cannot be represented with a single byte. When get() is used for SBCS, DBCS, or QBCS, each character is represented by its Unicode code point value.

```
MQERETURN osaMQeString_getSubstring(MQeExceptBlock* pExceptBlock,
                  MQEVOID*       pOutputBuffer,
                  MQEINT32*      pBufferLength,
                  MQETYPEOFSTRING requiredType,
                  MQECONST MQeStringHndl hString,
                  MQEINT32 from,
                  MQEINT32 to
                  );
```

This function is very similar to osaMQeString_get except that it only gets a substring (from **from** to **to** inclusive).

```
MQERETURN osaMQeString_getMQeSubstring(MQeExceptBlock* pExceptBlock,
                  MQeStringHndl *      phOutput,
                  MQECONST MQeStringHndl   hString,
                  MQEINT32 from,
                  MQEINT32 to
                  );
```

This function is very similar to osaMQeString_getSubstring except it returns its result as an MQeString.

```
MQERETURN osaMQeString_duplicate(MQeExceptBlock  * pExceptBlock,
                  MQeStringHndl * phNewString,
                  MQECONST MQeStringHndl   hString
                  );
```

This function duplicates an MQeString.

```
MQERETURN osaMQeString_codePointSize(MQeExceptBlock* pExceptBlock,
                  MQEINT32 * pSize,
                  MQECONST MQeStringHndl   hString
                  );
```

This function finds the memory size (in bytes) required for the largest character in the string.

```
MQERETURN osaMQeString_getCharLocation( MQeExceptBlock* pExceptBlock,
                  MQEINT32*       pOutIndex,
                  MQECONST MQeStringHndl    hString,
                  MQECHAR32       charToFind,
                  MQEINT32        startFrom,
                  MQEBOOL         searchForward
                  );
```

This function returns the location index (starting from 0) of the first appearance of a specified character, specified as its Unicode code point value. You can specify the starting point of your search and the direction of the search.

Tester

```
MQERETURN osaMQeString_isAsciiOnly(MQeExceptBlock*     pExceptBlock,
                       MQEBOOL*            pIsAsciiOnly,
                       MQECONST MQeString_* pString
                       );
```

This function determines whether the string contains any non-invariant ASCII characters.

```
MQERETURN osaMQeString_equalTo(MQeExceptBlock*   pExceptBlock,
                       MQEBOOL*          pIsEqual,
                       MQECONST MQeString_* pString,
                       MQECONST MQeString_* pEqualToString
                       );
```

This function determines whether two strings are equivalent.

```
MQERETURN osaMQeString_isNull(MQeExceptBlock  * pExceptBlock,
                       MQEBOOL * pIsNull,
                       MQECONST MQeStringHndl  hString
                       );
```

This function determines if a string is a null string. A a NULL handle is considered as a null string as well.

The Single Byte Character Set (SBCS) is the standard mode of operating with C on an ASCII code page. Java works in Unicode only and there may be platforms to support, that do not load an SBCS code page, for example in some countries languages are represented in DBCS. As it does not include the character pointer, the string item allows you to create strings on an ASCII machine without considering Unicode requirements. MQe carries out any necessary conversions. Use the UTF-8 representation of the string as this can cope with any character representation and does the conversion for you. Once created, an MQeString cannot be altered. However, a number of functions facilitate the use of the MQeString type. You can also create constant MQeStrings in a similar manner to using `#define NAME "mystring"`. Using MQeString ensures portability of the application.

Walkthrough: creating a basic application

This topic contains step-by-step instructions for creating a simple MQe application. It describes the steps you need to perform to create and configure your first queue manager, and then to verify that it can send and receive messages from another queue manager.

As well as describing what you need to do, it also tells you which MQe_Script commands you can use to perform each task simply. MQe_Script uses defaults for many attributes, which you would otherwise have to specify if you were writing equivalent code.

MQe_Script is available as a SupportPac from the IBM web site (see MQe SupportPacs) . The MQe_Script SupportPac includes full documentation on the use of all MQe_Script commands, including details of the defaults and explanations of how to change them if necessary.

You can also perform many of the steps involved in this process using the MQe_Explorer, which is another SupportPac available for download from the same web site.

Finally, the walkthrough provides links to pieces of example code that show you how to perform many of the steps programmatically.

Once a queue manager has been created and started, all of the configuration (including the creation of queues, connection definitions, remote queue definitions, and listeners) is performed using administration messages. For more information on the use of administration messages, see Configuration by messages overview.

1. Create a queue manager (QM1)

When you create a queue manager, you need to define the following attributes:

- Queue manager name
- Public or private registry
- Registry location
- Message store adapter
- Default queues
  - AdminQ
  - AdminReplyQ
  - DeadLetterQ

– System.default.local.Q

You can also set other (optional) attributes at this time, including a description, channel timeout, channel attribute rule name, and queue manager rule, but these are not included in this walkthrough.

For more information about the creation and configuration of queue managers, see Configuration by messages overview.

Creating QM1 using MQe_Script:

You can use the following MQe_Script command to create a queue manager called QM1:

```
mqe_script_qm -create -qmname QM1
```

This command creates a queue manager called QM1, with the following characteristics:

- Public registry
- A base location of C:\program files\mqe\java\mqe_script. The default registry and queue directories are in subdirectories in this path
- Uses the default message store and saves the information to disk
- Contains 4 default queues

An ini file is also created so that the queue manager information is saved and can be started again by passing the location of this file to an appropriate method.

Creating a queue manager programmatically:

For more information on using Java or C to create a queue manager, see “Creating a queue manager - step by step” on page 51. For examples in Java and C, see “Creating a simple queue manager in Java” on page 55 and “Creating a simple queue manager in C” on page 56.

2. Start the queue manager (QM1)

When you have created the queue manager called QM1, you need to start it.

Starting QM1 using MQe_Script:

You can use the following MQe_Script command to start the queue manager called QM1:

```
mqe_script_qm -load
```

When no name is supplied, this command starts the queue manager that has just been created. If you want to know how to load a queue manager and specify the INI file, see the documentation supplied with MQe_Script.

Starting a queue manager programmatically:

For more information on using Java or C to start a queue manager, see “Starting queue managers” on page 57. For examples in Java and C, see “Starting queue managers in Java” on page 57 and “Starting queue managers in C” on page 58.

3. Create a local queue (Q1)

When you have started the QM1 queue manager, you can create a local queue called Q1:

Creating Q1 using MQe_Script:

You can use the following MQe_Script command to create a local queue called Q1:

```
mqe_script_appq -create -qname Q1
```

This command creates a basic local queue called Q1, on the QM1 queue manager.

Creating a local queue programmatically:

For more information on using Java or C to create a local queue, see Configuring local queues. For examples in Java and C, see Java and C.

4. Create a connection definition

When you have created your local queue (Q1), you need to create a connection definition, specifying the following:

- The name of the queue manager that you want to connect to (the remote queue manager)
- The port on which the remote queue manager will be listening
- The communications adapter.

Creating a connection definition using MQe_Script:

You can use the following MQe_Script command to create a connection definition:

```
mqe_script_condef -create -cdname QM2 -port 1881
```

This command creates a connection definition to a queue manager called QM2, which is listening on port 1881. It is not necessary for QM2 to exist when the connection is created, but it must exist when you try to send a message to a remote queue on that queue manager. As no adapter is specified, the Http adapter is used by default.

Creating a connection definition programmatically:

For more information on using Java or C to create a connection definition, see Configuring connection definitions. For examples in Java and C, see Creating a connection definition (Java) and Creating a connection definition (C).

5. Create a remote queue definition

When you have created a connection definition, you need to create a remote definition of a local queue on queue manager QM2.

Creating a remote queue definition using MQe_Script:

You can use the following MQe_Script command to create a remote queue definition:

```
mqe_script_sproxyq -create -qname Q2 -destination QM2
```

This command creates a synchronous proxy queue, which is a remote definition of a local queue on QM2. It is not necessary for QM2 to exist when the remote queue definition is created. However, you must create a connection definition (see "4. Create a connection definition" on page 12) before you can create this remote queue definition.

Creating a remote queue definition programmatically:

For more information on using Java or C to create a remote queue definition, see Configuring remote queues. For examples in Java and C, see Java and C.

6. Create a listener (L1)

When you have created a remote queue definition, you need to create a listener.

Creating a listener using MQe_Script:

You can use the following MQe_Script command to create a listener called L1 (on queue manager QM1):

```
mqe_script_listen -create -listenname L1 -port 1882
```

Creates a listener for queue manager QM1 and listens on port 1882. The default communications adapter is used, which is the Http adapter.

Creating a listener programmatically:

For more information on using Java to create a listener, see Configuring a listener. For an example, see Java.

7. Start listener (L1)

When you have created a listener, you need to start it.

Starting a listener using MQe_Script:

You can use the following MQe_Script command to start the listener L1:

```
mqe_script_listen -start -listenname L1
```

Starting a listener programmatically:

For more information on using Java to start a listener, see Configuring a listener. For an example, see Java.

8. Create a second queue manager (QM2)

When you have finished configuring QM1 (as shown in the previous steps in this walkthrough), you need to create a second queue manager called QM2:

Creating QM2 using MQe_Script:

You can use the following MQe_Script command to create a queue manager called QM2:

```
mqe_script_qm -create -qmname QM2
```

This command creates a queue manager called QM2, with the following characteristics:

- Public registry
- A base location of C:\program files\mqe\java\mqe_script. The default registry and queue directories are in subdirectories in this path
- Uses the default message store and saves the information to disk
- Contains 4 default queues

An ini file is also created so that the queue manager information is saved and can be started again by passing the location of this file to an appropriate method.

Creating a queue manager programmatically:

To find out more about creating and configuring queue managers, see Configuration by messages overview. For more information on using Java or C to create a queue manager, see "Creating a queue manager - step by step" on page 51. For examples in Java and C, see "Creating a simple queue manager in Java" on page 55 and "Creating a simple queue manager in C" on page 56.

9. Start QM2

When you have created the queue manager called QM2, you need to start it.

Starting QM2 using MQe_Script:

You can use the following MQe_Script command to start the queue manager called QM2:

```
mqe_script_qm -load
```

When no name is supplied, this command starts the queue manager that has just been created. If you want to know how to load a queue manager and specify the INI file, see the documentation supplied with MQe_Script.

Starting a queue manager programmatically:

For more information on using Java or C to start a queue manager, see "Starting queue managers" on page 57. For examples in Java and C, see "Starting queue managers in Java" on page 57 and "Starting queue managers in C" on page 58.

10. Create a local queue (on QM2) called Q2

When you have started the QM2 queue manager, you can create a local queue called Q2.

Creating Q2 using MQe_Script:

You can use the following MQe_Script command to create a local queue called Q2:

```
mqe_script_appq -create -qname Q2
```

This command creates a basic local queue called Q2, on the QM2 queue manager.

Creating a local queue programmatically:

For more information on using Java or C to create a local queue, see Configuring local queues. For examples in Java and C, see Java and C.

11. Create a connection definition (on QM2)

When you have created your local queue (Q2), you need to create a connection definition, specifying the following:

- The name of the queue manager that you want to connect to (the remote queue manager)
- The port on which the remote queue manager will be listening
- The communications adapter.

Creating a connection definition using MQe_Script:

You can use the following MQe_Script command to create a connection definition:

```
mqe_script_condef -create -cdname QM1 -port 1882
```

This command creates a connection definition to a queue manager called QM1, which is listening on port 1882. It is not necessary for QM1 to exist when the connection is created, but it must exist when you try to send a message to a remote queue on that queue manager. As no adapter is specified, the Http adapter is used by default.

Creating a connection definition programmatically:

For more information on using Java or C to create a connection definition, see Configuring connection definitions. For examples in Java and C, see Creating a connection definition (Java) and Creating a connection definition (C).

12. Create a remote queue definition (on QM2)

When you have created a connection definition, you need to create a remote definition of a local queue on queue manager QM1.

Creating a remote queue definition using MQe_Script:

You can use the following MQe_Script command to create a remote queue definition:

```
mqe_script_sproxyq -create -qname Q1 -destination QM1
```

This command creates a synchronous proxy queue, which is a remote definition of a local queue on QM1. It is not necessary for QM1 to exist when the remote queue definition is created, but it must exist before a message is put to it.

Creating a remote queue definition programmatically:

For more information on using Java or C to create a remote queue definition, see Configuring remote queues. For examples in Java and C, see Java and C.

13. Create a listener (on QM2) called L2

When you have created a remote queue definition, you need to create a listener.

Creating a listener using MQe_Script:

You can use the following MQe_Script command to create a listener called L2 (on queue manager QM2):

```
mqe_script_listen -create -listenname L2 -port 1881
```

Creates a listener for queue manager QM2 and listens on port 1881. The default communications adapter is used, which is the Http adapter.

Creating a listener programmatically:

For more information on using Java to create a listener, see Configuring a listener. For an example, see Java.

14. Start the listener L2 (on QM2)

When you have created a listener, you need to start it.

Starting a listener using MQe_Script:

You can use the following MQe_Script command to start the listener L2:

```
mqe_script_listen -start -listenname L2
```

Starting a listener programmatically:

For more information on using Java to start a listener, see Configuring a listener. For an example, see Java.

15. Send (PUT) a message from QM1 to QM2

Now that you have created and started the two queue managers, created your queues and connection definitions, and created and started your listeners, you are in a position to send messages between the two queue managers.

Sending a message using MQe_Script:

On QM1, you can use the following MQe_Script command to send a message from QM1 to QM2:

```
mqe_script_msg -put -qname Q2 -qmname QM2
```

This command puts a message to queue Q2 on queue manager QM2.

Sending a message programmatically:

For more information on using Java or C to put a message to a queue, see “Messaging operations” on page 71. For examples in Java and C, see “Put message - assured put” on page 84 and “Put message - assured put” on page 84.

16. Receive (GET) the message on QM2

Now that a message has been put to the queue from QM1, you can get the message from QM2.

Receiving a message using MQe_Script:

You can use the following MQe_Script command to get the message from the queue:

```
mqe_script_msg -get -qname Q2 -qmname QM2
```

Receiving a message programmatically:

For more information on using Java or C to get a message from a queue, see "Messaging operations" on page 71. For examples, see "Get message - assured get" on page 88 and "Get message - assured get" on page 88.

17. Displaying details of MQe objects

You can display details of the MQe objects that you have created by issuing the `inquireall` MQe_Script command. For example, to see information about the local queue manager, use the following command:

```
mqe_script_qm -inquireall
```

This displays all the information about the local queue manager, and shows you any defaults that MQe_Script has used.

You can also display information about other objects, by specifying the object name. For example:

```
mqe_script_condef -inquireall -cdname QM2
```

An example MQe application (HelloWorld)

This topic describes how to create a basic application (called HelloWorld) using the MQe Java and C APIs. It contains information on designing, developing, deploying, and running the application.

Java ″HelloWorld″

This section describes how to design, develop, deploy, and run a basic ″HelloWorld″ application in Java.

Designing the Java application

This application aims to create and use a single queue manager with a local queue. It involves putting a message to the local queue and then removing it.

You can create queue managers for use by one program. Once this program has completed, you can run a second program that reinstates the previous queue manager configuration.

Typically, configuring new entities is a separate process from their actual use. Once configured, administering these entities also requires a different process than using them. This section concentrates on usage rather than administration.

Assuming that the queue manager entity has already been configured, the HelloWorld application has the following flow for both the C and Java codebases:

1. **Start the queue manager** This starts the queue manager based on information already created

2. **Create a message** Creates a structure that you can use to send a message from one queue manager to another
3. **Put to a local queue**Puts the message on the local queue
4. **Get from a local queue**Retrieves the message from the local queue and checks that the message is valid
5. **Shutdown**Clears and stops the queue manager

Developing the Java application

The following code is in the `examples.helloworld.Run` class in its complete state. Solutions using MQe classes are often separated into several separate tasks:

- Installation of the solution
- Configuration of the queue manager, leaving the configuration information on the local hard disk
- Use of the queue manager
- Removal of the queue manager
- Un-install of the solution

Before reading the information in this chapter, you need to configure a queue manager. The `examples.helloworld.Configure` program demonstrates the configuration of the queue manager. The `examples.helloworld.Unconfigure program` demonstrates the removal of the queue manager. This section of the documentation describes how to use the queue manager.

Overview of examples.helloworld.run:

The main method controls the flow of the hello world application. From this code, you can see that the queue manager is started, a message is put to a queue, a message is got from a queue, and the queue manager is stopped.

Trace information can be redirected to the standard output stream if the `MQE_TRACE_ON` symbolic constant has its' value changed to '`true`'.

```
public static void main(String[] args) {
        try {
            Run me = new Run();

            if (MQE_TRACE_ON) {
              me.traceOn();
            }
            me.start();
            me.put();
            me.get();
            me.stop();
              if (MQE_TRACE_ON) {
                me.traceOff();
            }
        } catch (Exception error) {
            System.err.println("Error: " + error.toString());
            error.printStackTrace();
        }
    }
```

Start the queue manager:

The `examples.helloworld.Configure` program creates an image of the `HelloWorldQM` queue manager on disk.

Before a queue manager can be used, it must be instantiated in memory, and started. The start method in the example program does this.

```
public void start() throws Exception {

      System.out.println("Starting the queue manager.");

        String queueManagerName = "HelloWorldQM";
        String baseDirectoryName =
          "./QueueManagers/" + queueManagerName;

        // Create all the configuration
      information needed to construct the
        // queue manager in memory.
        MQeFields config = new MQeFields();

       // Construct the queue manager section parameters.
        MQeFields queueManagerSection = new MQeFields();

        queueManagerSection.putAscii(MQeQueueManager.Name,
              queueManagerName);
        config.putFields(MQeQueueManager.QueueManager,
              queueManagerSection);

         // Construct the registry section parameters.
         // In this examples, we use a public registry.
        MQeFields registrySection = new MQeFields();

       registrySection.putAscii(MQeRegistry.Adapter,
          "com.ibm.mqe.adapters.MQeDiskFieldsAdapter");
        registrySection.putAscii(MQeRegistry.DirName,
              baseDirectoryName + "/Registry");

        config.putFields("Registry", registrySection);

        System.out.println("Starting the queue manager");
        myQueueManager = new MQeQueueManager();
        myQueueManager.activate(config);
        System.out.println("Queue manager started.");
    }
```

To start the queue manager, at a minimum you must know its name, location, and the adapter which should be used to read the queue manager's configuration information from its registry.

Activating the queue manager causes the configuration data from the disk to be read using the disk fields adapter, and the queue manager is then started and running, available for use.

Create a message and put to a local queue:

The following code constructs a message, adds a Unicode field with a value of ″Hello World!″ and the message is then put to the `SYSTEM.DEFAULT.LOCAL.QUEUE` on the local `HelloWorldQM` queue manager.

```
public void put() throws Exception {
        System.out.println("Putting the test message");
        MQeMsgObject msg = new MQeMsgObject();

    // Add my hello world text to the message.
        msg.putUnicode("myFieldName" , "Hello World!");
```

```
        myQueueManager.putMessage(queueManagerName,
        MQe.System_Default_Queue_Name, msg, null, 0L);
        System.out.println("Put the test message");
    }
```

Get message from a local queue: The following code gets the ″top″ message from the local queue, `SYSTEM.DEFAULT.LOCAL.QUEUE`, checks that a message with the field `myFieldName` was obtained, and displays the text held in the Unicode field.

```
public void get() throws Exception {
        System.out.println("Getting the test message.");
        MQeMsgObject msg = myQueueManager.getMessage( queueManagerName,
                                                      MQe.System_Default_Queue_Name,
                                                      null, null, 0L );

        if (msg != null) {
            System.out.println("Got the test message.");

            if (msg.contains("myFieldName")) {
                String textGot = msg.getUnicode("myFieldName");

                System.out.println("Message contained the text '" + textGot + "'");
            }
        }
    }
```

Stopping and deleting the queue manager:

This section describes how to stop a queue manager and delete the definition of the queue manager.

Stopping the queue manager

You can stop the queue manager using a controlled shutdown.

```
public void stop() throws Exception {
        System.out.println("Stopping the queue manager.");
        myQueueManager.closeQuiesce(QUIESCE_TIME);
        myQueueManager = null;
        System.out.println("Queue manager stopped.");
    }
```

Deleting the definition of the queue manager from the disk

You can use the`examples.helloworld.Unconfigure` program to remove the queue manager from disk.

Running the Java application

From a command prompt, set up your classpath to refer to the MQe class files. These are available in the Java directory, in which you installed the MQe product.

Ensure that your shell has the ability to create and modify the `./QueueManagers directory` on your system. If it does not have this ability, change the source of the `examples.helloworld programs`, such that they refer to an accessible directory, and re-compile the java code.

Invoke the Configure program to create the queue manager. The syntax depends on the Java Virtual Machine (JVM) you use. The IBM JVM is invoked using the ″java″ command, for example `java examples.helloworld.Configure`. This creates the queue manager on disk.

Run the `java examples.helloworld.Run` hello world program. This puts a message to a local queue, gets the message back and displays part of it.

You can now destroy the queue manager on the disk using `java examples.helloworld.Unconfigure`.

C ″HelloWorld″

This section describes how to design, develop, deploy and run a ″HelloWorld″ application in C.

Designing the C application

This application aims to create and use a single queue manager with a local queue. It involves putting a message to the local queue and then removing it.

You can create queue managers for use by one program. Once this program has completed, you can run a second program that reinstates the previous queue manager configuration.

Typically, configuring new entities is a separate process from their actual use. Once configured, administering these entities also requires a different process than using them. This section concentrates on usage rather than administration.

Assuming that the queue manager entity has already been configured, the HelloWorld application has the following flow for both the C and Java codebases:

1. **Start the queue manager** This starts the queue manager based on information already created
2. **Create a message** Creates a structure that you can use to send a message from one queue manager to another
3. **Put to a local queue**Puts the message on the local queue
4. **Get from a local queue**Retrieves the message from the local queue and checks that the message is valid
5. **Shutdown**Clears and stops the queue manager

Note: The C codebase does not have an equivalent of the Java Garbage Collection function. Therefore, clearing the queue manager features more strongly in C.

Developing the C application

This section covers the high level coding required for the ″HelloWorld″ application in C.

The code in the following examples is in the example HelloWorld_Runtime.c in its complete state. The example contains code to handle the specifics of running a program on a PocketPC, which mainly involves writing to a file to cope with the lack of command line options. Use the display function to write to a file, as shown in the examples contained in the following sections.

Overview of HelloWorld_Runtime.c:

You need to include just one header file to access the APIs. You must include the NATIVE definition to indicate that this is not the CBindings. You must also define the MQE_PLATFORM upon which you intend to run the application.

```
#define   NATIVE
#define  MQE_PLATFORM = PLATFORM_WINCE
#include<published/MQe_API.h>
```

All of the code, including variable declarations, is inside the main method. You require structures for error checking. The MQeExceptBlock structure is passed into all functions to get the error information back. In addition, all functions return a code indicating success or failure, which is cached in a local variable:

```
/* ... Local return flag */
 MQERETURN           rc;
 MQeExceptBlock      exceptBlock;
```

You must create a number of strings, for example for the queue manager name:

```
MQeStringHndl     hLocalQMName;

       ...

 if ( MQERETURN_OK == rc ) {
    rc = mqeString_newUtf8(&exceptBlock,
            &hLocalQMName,
            "LocalQM");
}
```

The first API call made is session initialize:

```
/* ... Initalize the session */
rc = mqeSession_initialize(&exceptBlock);
```

Start the queue manager:

This process involves two steps:

1. Create the queue manager item.
2. Start the queue manager.

Creating the queue manager requires two sets of parameters, one set for the queue manager and one for the registry. Both sets of parameters are initialized. The *queue store* and the registry require directories.

Note: All calls require a pointer to ExceptBlock and a pointer to the queue manager handle.

```
 if (MQERETURN_OK == rc) {

MQeQueueManagerParms qmParams  = QMGR_INIT_VAL;
MQeRegistryParms     regParams = REGISTRY_INIT_VAL;
qmParams.hQueueStore           = hQueueStore;
qmParams.opFlags               = QMGR_Q_STORE_OP;

/* ... create the registry parameters -
    minimum that are required */
regParams.hBaseLocationName     =  hRegistryDir;
display("Loading Queue Manager from registry \n");
rc = mqeQueueManager_new(  &exceptBlock,
                        &hQueueManager,
                         hLocalQMName,
                         &qmParams,
                         &regParams);
}
```

You can now start the queue manager and carry out messaging operations:

```
 /* Start the queue manager  */

if ( MQERETURN_OK == rc ) {
   display("Starting the Queue Manager\n");
   rc = mqeQueueManager_start(hQueueManager,
          &exceptBlock);
}
```

Create a message:

To create a message, firstly create a new fields object. The following example adds a single field. Note that the field label strings are passed in:

```
MQeFieldsHndl hMsg;

display("Creating a new message\n");
rc = mqeFields_new(&exceptBlock,&hMsg);
if ( MQERETURN_OK == rc ) {
  rc = mqeFields_putInt32(hMsg,&exceptBlk,
        hFieldLabel,42);
}
```

Put message to a local queue:

Once you have created the message, you can put it to a local queue using the *putMessage* function. Note that the queue and queue manager names are passed in. NULL and 0 are passed in for the security and assured delivery parameters, as they are not required in this example. Once the message has been put, you can free the MQeFields object:

```
if ( MQERETURN_OK == rc ) {
      display("Putting a message \n");
       rc = mqeQueueManager_putMessage(hQueueManager,
                                     &exceptBlock,
                                      hLocalQMName,
                                      hLocalQueueName,
                                     hMsg,
                                      NULL,
                                      0);

     (void) mqeFields_free(hMsg,NULL);
  }
```

Get message from a local queue:

Once the message has been put to a queue, you can retrieve and check it. Similar options are passed to the getMessage function. The difference is that a pointer to a field’s handle is passed in. A new Fields object is created, removing the message from the queue:

```
MQeFieldsHndl hReturnedMessage;
display("Getting the message back \n");

rc = mqeQueueManager_getMessage(hQueueManager,
                      &exceptBlock,
                      &hReturnedMessage,
                      hLocalQMName,
                      hLocalQueueName,
                            NULL,
                            NULL,
                            0);
}
```

Once the message has been obtained, you can check it for the value that was entered. Obtain this by using the getInt32 function. If the result is valid, you can print it out:

```
if (MQERETURN_OK == rc) {
  MQEINT32 answer;
   rc = mqeFields_getInt32(hReturnedMessage,
                           &exceptBlock,
                           &answer,
                           hFieldLabel);

if (MQERETURN_OK == rc) {
  display("Answer is %d\n",answer);
}
else {
   display("\n\n %s (0x%X) %s (0x%X)\n",
       mapReturnCodeName(EC(&exceptBlock)),
       EC(&exceptBlock),
       mapReasonCodeName(ERC(&exceptBlock)),
       ERC(&exceptBlock)  );
  }

}
```

Shutdown:

Following the removal of the message from the queue, you can stop and free the queue manager. You can also free the strings that were created. Finally, terminate the session:

```
(void)mqeQueueManager_stop(hQueueManager,&exceptBlock);
(void)mqeQueueManager_free(hQueueManager,&exceptBlock);

/* Lets do some clean up */
(void)mqeString_free(hFieldLabel,&exceptBlock);
(void)mqeString_free(hLocalQMName,&exceptBlock);
(void)mqeString_free(hLocalQueueName,&exceptBlock);
(void)mqeString_free(hQueueStore,&exceptBlock);
(void)mqeString_free(hRegistryDir,&exceptBlock);


(void)mqeSession_terminate(&exceptBlock);
```

Compiling:

To simplify the process of compiling, the examples directory includes a makefile. This is the makefile exported from eMbedded Visual C (EVC). A batchfile runs this makefile. This batch file will setup the paths to the EVC directories, along with the paths to the MQe installation. You might need to edit the batch file, depending on how you want to install MQe.

Running the batch file compiles the example. Bye default, the batch file compiles for Debug PocketPC 2000 (either Emulator or ARM processor).

Deploying the C application

In order to deploy the ″HelloWorld″ application, you need to create a queue manager. There are various ways to do this, which are covered elsewhere in this information center. In this case, the HelloWorld_Admin program is used. Run this as described below.

The following instructions are applicable to both the emulator and an actual device:

1. Copy across all the DLLs to the root of the device. Take these from either the arm or x86 emulator directories.
2. Build the example code using the supplied makefile.

   Note: You need to compile the HelloWorld_Admin.c and HelloWorld_Runtime.c files.
3. Copy across these binaries to the device or emulator that is running PocketPC or Emulator.

Running the C application

This section describes how to run the ″HelloWorld″ application in Java and C, on the PocketPC or emulator.

This example involves two steps:
1. Create the queue manager. To do this, run the HelloWorld_Admin program.Running this creates the persistent disk representation of the QueueManager.
2. Run the HelloWorld_Runtime program. This starts a QueueManager based upon the established registry. To check the program has worked correctly, look at the log file that has been generated. By default, this is in the root of the device.

Using the MQe development and administration tools

The following are some of the tools that you can use to develop or administer MQe applications:

MQe_Explorer
: The MQe_Explorer provides a graphical user interface for the management of an MQe network and its interconnection with WebSphere MQ. It allows MQe queue managers and their associated objects, such as queues, connections, and bridges, to be locally or remotely configured. MQe_Explorer also provides a simple way of creating local queue managers, which can then be further configured to meet the needs of applications. It also offers a launch and debug environment for MQe applications. MQe_Script is available as a SupportPac that you can download from the WebSphere MQ Everyplace web site. For more information see MQe SupportPacs.

MQe_Script
: MQe_Script is a command-line based tool for MQe, and is platform independent. It allows MQe queue managers and their associated objects, such as queues, connections, listeners, and bridge objects to be locally or remotely configured. Test messages can also be sent to the queues to validate the operation of the network. Like the MQe_Explorer, MQe_Script provides a simple way of creating local queue managers, which you can then configure and extend for use by your application. MQe_Explorer is available as a SupportPac that you can download from the WebSphere MQ Everyplace web site. For more information see MQe SupportPacs.

WebSphere Studio Application Developer
: WebSphere Studio Application Developer is an integrated development environment for visually designing, constructing, testing, and deploying Web services, portals, and Java 2 Enterprise Edition (J2EE) applications. It is built on Eclipse, and provides templates, wizards, and drag-and-drop

tools that allow you to create Java applications quickly and simply. For more information on WebSphere Studio Application Developer, see:

`http://www.ibm.com/software/awdtools/studioappdev`

WebSphere Studio Device Developer
WebSphere Studio Device Developer provides an integrated development environment (IDE) for building, testing, and deploying Java 2 Micro Edition (J2ME) applications that run on wireless devices such as cellular telephones, personal digital assistants (PDA), and handheld computers. For more information on WebSphere Studio Device Developer, see:

`http://www.ibm.com/software/wireless/wsdd/`

Eclipse
Eclipse is an open industry-supported platform for software development tools. It provides a plug-in based framework that facilitates the creation, integration, and use of software tools. For more information on Eclipse, see:

`http://www.eclipse.org`

Using WebSphere Studio Device Developer (WSDD)

This topic describes how to develop and deploy applications to devices from WebSphere Studio Device Developer (WSDD). To fully understand the concepts outlined here, you should have Java programming skills, knowledge of J2ME and MIDlets, and basic knowledge of MQe.

The example application aims to aid your understanding of the MQe interface. The code can be split into 3 parts:

The message service
This runs MQe, controls a queue manager and performs functions such as queue creation and message sending. This is the core of the examples and allows them to be written with minimal calls to the MQe API. This also means that to see the code required to create a local queue for example, a user can simply look at the relevant function within MQeMessageService.

Example 1: The message pump
This is a very simple application consisting of a single server and client. The client is set to send a message to the server every 3 seconds which, when received by the server, will be displayed to the user. Queues are asynchronous. Implementations of the client are available for both MIDP and J2SE, while the server is only available for J2SE.

Example 2: The text application
This is slightly more complex than the first example, consisting of 2 servers and a client. When initiating, the client is required to register with the registration server. The registration server adds the client to a store-and-forward queue on the gateway server and replies with a success or failure message. The client can then send user-defined messages to the gateway server (which it will display). The aim of this application is to show how a separate server can be used to create resources necessary for a new client on the system to aid scalability of large MQe networks.

Developing applications for Palm

This topic explains how to set up the Palm device and WebSphere Studio Device Developer (WSDD) to work together.

Palm: What you need to get started

The following are prerequisites required for writing and testing applications for the Palm:

- A Palm device or Palm Emulator (you can download POSE from `http://www.palmos.com/dev/tools/emulator/`)
- A copy of a J2ME virtual machine installed on the Palm, for example the Sun's K Virtual Machine (KVM) and IBM's J9, available from `http://java.sun.com` and `http://www.embedded.oti.com`
- A cradle to synchronize the palm with your PC
- Something to generate .prc files, that is Palm executables, to run on the Palm, such as Sun's J2ME Wireless Toolkit (available at `http://java.sun.com`) and IBM's WSDD, available at `http://www.embedded.oti.com`, which includes J9 as standard
- MQe JARs/classes

This documentation concentrates on J9 and WSDD.

Palm: Getting started with WSDD

You must complete several tasks before using the Palm device or Palm emulator to run MQe MIDlets:

1. Install the virtual machine onto the unit. The .prc files required for this are located in the C:\IBM\wsdd\wsdd4.0\ive\runtimes\palmos\68k\ive\bin directory or the equivalent location for your installation. You need the following files:
   - j9_vm_bundle.prc
   - j9pref.prc
   - midp15.prc
   - j9_dbg_bundle.prc (only if you are planning to debug an application)
2. Once you have installed these files on your palm device (it should come with instructions on how to do this), use WSDD to create a new MIDlet suite (in the Java perspective - [File][New][Other][J2ME for J9][Create MIDlet Suite]).
3. Import the source for the example application into the src directory. Include the MQe library in the list of libraries to use, that is right-click the name of the project in the packages window and select [Properties] [Java Build Path] and the Libraries tab. Use the 'Add External JARs' option to add the MQe MIDP jar to the list. Note the following files are not meant for use under MIDP:
   - mqeexampleapp.msgpump.NormalClient
   - mqeexampleapp.msgpump.NormalServer
   - mqeexampleapp.msgpump.InputThread
   - mqeexampleapp.textapp.Client
   - mqeexampleapp.textapp.GatewayServer
   - mqeexampleapp.textapp.RegistrationServer

   These should be run in Foundation or J2SE to act as command-line implementations of the clients and servers. There are no MIDP servers as it is not an environment that servers are designed to run in.
4. Set WSDD to run files on that device. With normal Palms, an installation program is provided to enable the installation of new programs from a desktop computer (e.g. C:\Palm\Instapp.exe). This needs to be set in WSDD in

[Window][Preferences][Device Developer][PalmOS Java Configuration] under PalmOS Install Tool. You also meed to set the other options in this menu:

PalmOS Emulator
This is required if you want to use POSE or a similar PalmOS emulator

PilRC resource compiler
This creates the PRC files from the jad and jar. The WSDD help describes the java options in more detail.

Palm: Building for the Palm in WSDD

Once WSDD has been set up to work with the palm, try building and running the example application on your palm device:

1. Double-click the wsddbuild.xml file from within your project. If you created a J2ME for J9 project and not a normal Java one, it will appear after all the packages.
2. Select the builds tab from the bottom of the window. Currently, your list of builds should be empty. This window specifies the platforms you are building the project for, that is Palm, PocketPC, Windows, and so on.
3. Click **Add Build** and select the palm option from the pulldown platforms menu.
4. Click **Next** and enter any creator ID and a name for the application.
5. Click **Next** again until you reach the final *select launcher* screen. If you are using a palm device, select the manual option. If you are using the emulator, select the emulator option.
6. Click **Finish** and select the launch tab. Your device should now be a launch option.

Developing applications for PocketPC

This topic explains how to set up the PocketPC device and WebSphere Studio Device Developer (WSDD) to work together.

PocketPC: What you need to get started

To run MQe applications on the PocketPC you need:

- A Pocket PC device. Emulators exist, but they are not as true to the original device as the Palm emulators are to the Palm.A copy of a J2ME virtual machine installed on the device. A cradle to sync the PocketPC with your desktop
- J9 for the PocketPC comes with WSDD and is located in C:\IBM\wsdd\wsdd4.0\ive\runtimes\pocketpc\arm\ive. The files required from here are:
  - bin\iverel15.dll
  - bin\j9.exe
  - bin\j9dbg15.dll
  - bin\j9dyn15.dll
  - bin\j9hook15.dll
  - bin\j9midp15.dll
  - bin\j9prt15.dll
  - bin\j9thr15.dll
  - bin\j9vm15.dll
  - bin\j9w.exe

– bin\j9zlib15.dll
– bin\swt-win32-ce-2023.dll
– lib\jclMidp
– lib\jclMidp.jxe

These are specified in the WSDD help file. Create a similar directory structure on the device, for example, program files or WSDD with bin and lib subdirectories. Then copy the files to the relevant places. Note that the example application functions under MIDP, hence the need for the jclMidp.jxe file. The section *Palm: What you need to get started* in "Developing applications for Palm" on page 26 provides details on downloading WSDD.

PocketPC: Getting started with WSDD

To run applications on the PocketPC from WSDD, you need to tell WSDD where the various files you copied to your device are located. This is done in [Window][Preferences][Device Developer][PocketPC Java Configuration]. Set the three options to: \Program Files\WSDD \My Documents\WSDD \Windows\Start Menu, assuming that you copied the J9 files to '\Program Files\WSDD' earlier

PocketPC: Building for the Pocket PC in WebSphere Studio Device Developer

This procedure is almost identical to that described in the Building for the Palm in WSDD section. However, with the final choice for launcher, choose 'MIDlet Suite on PocketPC Device' rather than the manual option. This means that the application automatically copies to the relevant device and runs automatically.

Debugging applications

This section describes how to set the example application debugging using WSDD, both locally and remotely, on various devices.

Debugging on the Palm using WSDD

To debug on the palm using WSDD:
1. Install the j9_dbg_bundle.prc file on the target device before attempting to debug. This is located in C:\IBM\wsdd\wsdd4.0\ive\runtimes\palmos\68k\ive\bin.
2. On on the target device, run prefs and navigate to the J9 Java VM section. Ensure that 'Enable Debug' is selected, otherwise you cannot debug an application.
3. When the application launches (via wsddbuild.xml - launches), select **debug** rather than **run**.

Debugging on the PocketPC using WSDD

The files specified in the *PocketPC: What you need to get started* section include the necessary components to debug remotely. Simply launch the application using the debug command rather than the run command, as described in *Debugging on the palm using WSDD*.

Debugging locally using WSDD

Select **debug** rather than **run** to start the application.

Runnable classes

The following classes can be run the prompt:

mqeexampleapp.msgpump.NormalClient
A J2SE client for the Message Pump

mqeexampleapp.msgpump.NormalServer
A J2SE server for the Message Pumpv

mqeexampleapp.textapp.Client
A J2SE client for the Text App

mqeexampleapp.textapp.RegServer
A J2SE registration server for the Text App

mqeexampleapp.textapp.GatewayServer
A J2SE gateway server for the Text App

MIDlets

The following MIDlets are available in this example application:

mqeexampleapp.msgpump.MidpClient
The MIDP client for the Message Pump

mqeexampleapp.textapp.MidpClient
The MIDP client for the Text App

mqeexampleapp.messageservice.RMSclea
A simple utility to clear all RMS stores within the MIDlet suite

Giving parameters to the MIDlet

A useful feature of MIDlets is that they can retrieve parameters from their jad file. The example applications take advantage of this to allow simple changes to the MIDP clients without having to alter the code. Unfortunately, you cannot perform the necessary changes to the jad file in WSDD. To use this feature, open the jad of your project in a text editor. User defined parameters are specified as follows:

```
parameter: value
```

Use the following parameters for the two example applications:

1. MsgPump Client

   Pump_SecurityLevel
   Specifies the security level that the application should use:
   - 0 for no security
   - 1 for message based security
   - 2 for queue based security

   Pump_ServerQueue
   Specifies the name of the queue that messages should be sent to

   Pump_ServerIP
   Specifies the IP of the server that will sent messages

   Pump_ServerPort
   Specifies the port that the server will be listening on

   Pump_ServerQueueManager
   Specifies the name of the queue manager that messages will be sent to

2. TextApp Client

   pp_Registration_ServerIP
   The IP address of the registration server

   App_Gateway_ServerIP
   The IP address of the gateway server

An example jad file may look something like this:

```
MIDlet-Version:
MIDlet-Name: ExampleAppv2NewestMQe MIDlet-Jar-Size:
MIDlet-Jar-URL:
MIDlet-1: pumpclient,,mqeexampleapp.msgpump.MidpClient
MIDlet-2: clear,,mqeexampleapp.messageservice.RMSclear
MIDlet-3: textapp,,mqeexampleapp.textapp.MidpClient
MIDlet-Vendor:

Pump_SecurityLevel: 0
Pump_ServerQueue: A_Queue
Pump_ServerPort: 8083
Pump_ServerQueueManager: QM_Mr_Server
Pump_ServerIP: 10.0.0.101

App_Registration_ServerIP: 10.0.0.100
App_Gateway_ServerIP: 10.0.0.131
```

Any value not specified in this manner defaults to its usual value.

Cleaning up after applications

As with all MQe queue managers, registries and messages are left on the system after the queue manager has been shut down. This design allows queue managers to restart without losing their messages or recreating all their queues and registry settings. If one of the examples crashes on starting, the data they leave behind should automatically be removed to prevent them from being restarted with an incomplete registry. If the example does not crash, or you wish to start the queue manager from scratch, you can use the following methods to remove the registry from the system:

J2SE The example application uses the MQeDiskFieldsAdapter, which will save registry settings to the Hard Drive. These are located in c:/MQe/QM/. Delete this directory to remove the remaining information.

MIDP The example application uses the MQeMidpFieldsAdapter for MIDP environments. This means that the registry is stored in the record stores of the MIDlet Suite. You can remove them using the RMSclear MIDlet located in the exampleapp.messageservice package.

Note: If you delete a MIDlet Suite from a device, its record store is also removed.

Constraints of SmartLinker

A program called SmartLinker is used to strip unused classes and methods before packing a project into a .jxe file. Although this gives the benefit of a much smaller application, it also causes dynamically loaded classes to be stripped from the application when the .jxe is built.

An example of this is the various adapters that are dynamically loaded for different environments. Because these adapters are not explicitly referred to anywhere in the code, they are removed and so a NoClassDefFoundException is thrown.

The cleanest way to solve this problem is to specify in the jxeLinkOptions file that you wish to include a specific class. You can do this in WSDD in the following manner:

1. In the packages view of your project, open the directory for the device you are creating the jxe for (e.g. palm68k) and open the jxeLinkOptions file (e.g. ExampleApp.jxeLinkOptions.
2. Select the in or exclusion tab and pic [include whole classes] from the pulldown menu. This screen shows all the classes that the user has specified will definitely be included.
3. To add a new class to the list, select [new] and enter the class in the [Rule pattern] box, for example . com.ibm.mqe.adapter.MQeMidpFieldsAdapter. The following files require inclusion in this manner for the MIDP clients to work:
   - mqeexampleapp.messageservice.QueueManagerRules
   - com.ibm.mqe.adapters.MQeMidpFieldsAdapter
   - com.ibm.mqe.adapters.MQeMidpHttpAdapter
   - com.ibm.mqe.MQeAttributeRule
   - com.ibm.mqe.messagestore.MQeMessageStore
   - com.ibm.mqe.registry.MQeFileSession

Further information

WSDD help provides additional information under: Websphere Studio Device Developer Product Documentation\WSDD Product Documentation \Tasks\Working with Palm OS Targets

Designing your real application

Messaging

Overview of MQe messaging

The MQe programming model uses several entities, for example messages, queues, and queue managers, that work together as a flexible toolkit. Each entity has a specific purpose and works together with other entities to provide solutions for message topologies.

What are MQe messages?

Introduction to the use of MQe messages

Messages are collections of data sent by one application and intended for another application. MQe messages contain application-defined content. When stored, they are held in a queue and such messages may be moved across an MQe network.

MQe messages are a special type of MQeFields items, as described in "MQeFields" on page 38. Therefore, you can use methods that are applicable to MQeFields with messages.

Therefore, messages are Fields objects with the addition of some special fields. Java provides a subclass of MQeFields, MQeMsgObject which provides methods to manage these fields. The C codebase does not provide such a subclass. Instead, there are a number of mqeFieldsHelper_operation functions. The following fields form the `Unique ID` of an MQe message:

- In Java, the timestamp, generated when the message is first created or, in C, when the message is first put to a queue
- The name of the queue manager, to which the message is first put.

The `Unique ID` identifies a message within an MQe network provided all queue managers within the MQe network are named uniquely. However, MQe does not check or enforce the uniqueness of queue manager names.

In Java, the message is created when an instance of MQeMsgObject is created. In C, the Message is "created", that is UniqueID fields are added, when the message is put to a queue.

The getMsgUIDFields()method or mqeFieldsHelpers_getMsgUidFields() function accesses the `UniqueID` of a message, for example:

Java code

```
MQeFields msgUID = msgObj.getMsgUIDFields();
```

C code

```
rc = mqeFieldsHelpers_getMsgUidFields(hMgsObj,
                 &exceptBlock,&hUIDFields);
```

MQe adds property related information to a message (and subsequently removes it) in order to implement messaging and queuing operations. When sending a message between queue managers, you can add resend information to indicate that data is being retransmitted.

© Copyright IBM Corp. 2000, 2004

Typical application-based messages have additional properties in accordance with their purpose. Some of these additional properties are generic and common to many applications, such as the name of the reply-to queue manager.

Message properties

Table of MQe message properties

MQe supports the following message properties:

Table 1. Message properties

Property name	Java type	C type	Description
Action	int	MQEINT32	Used by administration to indicate actions such as inquire, create, and delete
Correlation ID	byte[]	MQEBYTE[]	Byte string typically used to correlate a reply with the original message
Errors	MQeFields	MQeFieldsHndl	Used by administration to return error information
Expire time	int or long	MQEINT32 or MQEINT64	Time after which the message can be deleted (even if it is not delivered)
Lock ID	long	MQEINT64	The key necessary to unlock a message
Message ID	byte[]	MQEBYTE[]	A unique identifier for a message
Originating queue manager	string	MQeStringHndl	The name of the queue manager that sent the message
Parameters	MQeFields	MQeFieldsHndl	Used by administration to pass administration details
Priority	byte	MQEBYTE	Relative order of priority for message transmission
Reason	string	MQeStringHndl	Used by administration to return error information
Reply-to queue	string	MQeStringHndl	Name of the queue to which a message reply should be addressed
Reply-to queue manager	string	MQeStringHndl	Name of the queue manager to which a message reply should be addressed
Resend	boolean	MQEBOOL	Indicates that the message is a resend of a previous message
Return code	byte	MQEBYTE	Used by administration to return the status of an administration operation
Style	byte	MQEBYTE	Distinguishes commands from request/reply for example
Wrap message	byte[]	MQEBYTE[]	Message wrapped to ensure data protection

Symbolic names:

Table of symbolic names corresponding to MQe message properties

The following table lists the symbolic names corresponding to the MQe message properties:

Table 2. Symbolic names that correspond to message property names

Property name	Java constant	C constant
Action	MQeAdminMsg.Admin_Action	MQE_ADMIN_ACTION
Correlation ID	MQe.Msg_CorrelID	MQE_MSG_CORRELID
Errors	MQeAdminMsg.Admin_Errors	MQE_ADMIN_ERRORS
Expire time	MQe.Msg_ExpireTime	MQE_MSG_EXPIRETIME
Lock ID	MQe.Msg_LockID	MQE_MSG_LOCKID
Message ID	MQe.Msg_MsgID	MQE_MSG_MSGID
Originating queue manager	MQe.Msg_OriginQMgr	MQE_MSG_ORIGIN_QMGR
Parameters	MQeAdminMsg.Admin_Params	MQE_ADMIN_PARAMS
Priority	MQe.Priority	MQE_MSG_PRIORITY
Reason	MQeAdminMsg.Admin_Reason	MQE_ADMIN_REASON
Reply-to-queue	MQe.Msg_ReplyToQ	MQE_MSG_REPLYTO_Q
Reply-to queue manager	MQe.Msg_ReplyToQMgr	MQE_MSG_REPLYTO_QMGR
Resend	MQe.Msg_Resend	MQE_MSG_RESEND
Return code	MQeAdminMsg.Admin_RC	MQE_ADMIN_RC
Style	MQe.Msg_Style	MQE_MSG_STYLE
Wrap message	MQe.Msg_WrapMsg	MQE_MSG_WRAPMSG

Examples:

Message Properties - Examples

In all cases, a defined constant allows the property name to be carried in a single byte. For example, priority (if present) affects the order in which messages are transmitted, correlation ID triggers indexing of a queue for fast retrieval of information, expire time triggers the expiry of the message, and so on. Also, the default message dump command minimizes the size of the generated byte string for more efficient message storage and transmission.

The MQe *Message ID* and *Correlation ID* allow the application to provide an identity for a message. These are also used in interactions with the rest of the MQ family:

Java

```
MQeMsgObject msgObj = new MQeMsgObject;
msgObj.putArrayOfByte( MQe.Msg_ID, MQe.asciiToByte( "1234" ));
```

C

```
rc = mqeFields_putArrayOfByte(hMsg,&exceptBlock,
          MQE_MSG_MSGID,pByteArray,sizeByteArray);
```

Priority contains message priority values. Message priority is defined as in other members of the MQ family. It ranges from 9 (highest) to 0 (lowest):

Java

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putByte( MQe.Msg_Priority, (byte)8 );
```

C

```
rc = mqeFields_putByte(hsg,&exceptBlock, MQE_MSG_PRIORITY, (MQEBYTE)8);
```

Applications can create fields for their own data within messages:

Java

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "PartNo", "Z301" );
msgObj.putAscii( "Colour", "Blue" );
msgObj.putInt( "Size", 350 );
```

C

```
MQeFieldsHndl hPartMsg;
MQeStringHndl hSize_FieldLabel;
rc = mqeFields_new(&exceptBlock,&hPartMsg);
rc = mqeString_newUtf8(&exceptBlock,
                 &hSize_FieldLabel,"Size");

rc = mqeFields_putInt32(hPartMsg,
                &exceptBlock,hSize_FieldLabel,350);
```

The priority of the message is used, in part, to control the order in which messages are removed from the queue. If the message does not specify any, then the queue default priority is used . This, unless changed, is 4. However, the application must interpret the different levels of priority.

In Java, you can extend the MQeMsgObject to include some methods that assist in creating messages, as shown in the following example:

```
package messages.order;
import com.ibm.mqe.*;

/*** This class defines the Order Request format */
public class OrderRequestMsg extends MQeMsgObject
{

  public OrderRequestMsg() throws Exception
  {
  }

 /*** This method sets the client number */
  public void setClientNo(long aClientNo) throws Exception
  {
    putLong("ClientNo", aClientNo);
  }

 /*** This method returns the client number */
  public long getClientNo() throws Exception
  {
    return getLong("ClientNo");
  }
```

To find out the length of a message, you can enumerate on the message as each data type has methods for getting its length.

Message filters

Introduction to MQe message filters

Filters allow MQe to perform powerful message searches. Most of the major queue manager operations support the use of filters. You can create filters using MQeFields.

Using a filter, for example in a getMessage() call, causes an application to return the first available message that contains the same fields and values as the filter. The following examples create a filter that obtains the first message with a message id of ″1234″:

Java

```
MQeFields filter = new MQeFields();
filter.putArrayOfByte( MQe.Msg_MsgID,
       MQe.AsciiToByte( "1234" ) );
```

C
rc = mqeFields_putArrayOfByte(hMsg,, &exceptBlock, MQE_MSG_MSGID, pByteArray, sizeByteArray);

You can use this filter as an input parameter to various API calls, for example `getMessage`.

Message expiry

Overview of the expiry of messages in queues

Queues can be defined with an expiry interval. If a message has remained on a queue for a period of time longer than this interval then the message is automatically deleted. When a message is deleted, a queue rule is called. Refer to the Rules topic for information on queue rules. This rule cannot affect the deletion of the message, but it does provide an opportunity to create a copy of the message.

Messages can also have an expiry interval that overrides the queue expiry interval. You can define this by adding a C `MQE_MSG_EXPIRETIME` or Java `MQe.Msg_ExpireTime` field to the message. The expiry time is either relative (expire 2 days after the message was created), or absolute (expire on November 25th 2000, at 08:00 hours). Relative expiry times are fields of type Int or MQEINT32, and absolute expiry times are fields of type Long or MQEINT64.

In the example below, the message expires 60 seconds after it is created (60000 milliseconds = 60 seconds).

```
/* create a new message          */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
/* expiry time of sixty seconds after message was created    */
msgObj.putInt( MQe.Msg_ExpireTime, 60000 );
```

In the example below, the message expires on 15th May 2001, at 15:25 hours.

```
/* create a new message          */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
/* create a Date object for 15th May 2001, 15:25 hours      */
Calendar calendar = Calendar.getInstance();
calendar.set( 2001, 04, 15, 15, 25 );
Date expiryTime = calendar.getTime();
/* add expiry time to message       */
msgObj.putLong( MQe.Msg_ExpireTime, expiryTime.getTime() );
/* put message onto queue       */
qmgr.putMessage( null, "MyQueue", msgObj, null, 0 );
```

Checking for expired messages:

Explanation of when MQe checks for expired messages

A message is checked for expiry when:

It is added to a queue
: Expiry can occur when a message is added from the local API, pulled down via a Home Server Queue, or pushed to a queue.

It is removed from a queue
: Expiry can occur when a message can be removed from the local API, or when a message is pulled remotely.

A queue is activated
: When a queue is activated, a reference to the queue is created in memory. Any message that has expired is removed. The state of the message is irrelevant to this operation.

A queue is deleted
: If an admin message arrives to delete a queue, the queue must be empty first. Therefore, before this check is done, any expired messages are removed from the queue. The state of the message is irrelevant to this operation.

A queue is checked for size
: If an admin message arrives to inquire on the size of a queue, the queue is first purged of admin messages.

You can add a queue rule to notify you when messages expire. However, in a certain situation between two queue managers, a message may seem to expire twice. This is not because the message has been duplicated, but is outlined in the following paragraph.

Assume that an asynchronous queue has a message on it due to expire at 10:00 1st Jan 2005. All messages on such queues are transmitted using a 2 stage process. This process is equivalent to a putMessage and confirmPutMessage pair of operations. Suppose that the first transmission stage occurs at 09:55. A reference to the message appears on the remote queue manager. However, it is not yet available to an application on that queue manager. Then, if the network fails until 10:05, the expiry time of the message is missed. Therefore, the message expires on the remote queue and the queue expiry rule gets fired. Also, in due course, the queue expiry rule gets fired on the destination queue manager.

Assurance of expiry:

Explains how to ensure message expiry

The expiry time can be calculated to the millisecond. For correct operation the clocks of the machines running the queue managers must be accurately aligned. Failure to do this within accuracy determined by your choice of expiry times causes messages to appear active on one queue manager, while they have expired on others. Ensure that you use the correct field type for the expiry value. An int (32 bit) field is used for relative expiry times, and a long (64 bit) field is used for absolute times. The field name is the same in both cases.

MQeFields

Overview of the MQeFields container structure

MQeFields is a container data structure widely used in MQe. You can put various types of data into the container. It is particularly useful for representing data that needs to be transported, such as messages. The following code creates an MQeFields structure:

Java code

```
/* create an MQeFields  object      */
MQeFields fields = new MQeFields( );
```

C code

```
MQeFieldsHndl hFields;
  rc = mqeFields_new(&exceptBlock, &hFields);
```

MQeFields contains a collection of orderless fields. Each field consists of a triplet of entry name, entry value, and entry value type. MQeFields forms the basis of all MQe messages.

Use the entity name to retrieve and update values. It is good practice to keep names short, because the names are included with the data when the MQeFields item is transmitted.

The name must:
- Be at least 1 character long
- Conform to the ASCII character set (characters with values 20 < value < 128)
- Exclude any of the characters { } [ ] # ( ) : ; , ' " =
- Be unique within MQeFields

Storage and retrieval of values in MQeFields

Examples of storing values in an MQeFields item, and retrieving values from an MQeFields item

The following example shows how to store values in an MQeFields item:

Java code

```
/* Store integer values into a fields object    */
  fields.putInt( "Int1", 1234 );
  fields.putInt( "Int2", 5678 );
  fields.putInt( "Int3",    0 );
```

C code

```
MQeStringHndl hFieldName;
  rc = mqeString_newChar8(&errStruct,  &hFieldName, "A Field Name");
  rc = mqeFields_putInt32(hNewFields,&errStruct,hFieldName,1234);
```

The following example shows how to retrieve values from an MQeFields item:

Java code

```
/* Retrieve an integer value from a fields object    */
  int Int2 = fields.getInt( "Int2" );
```

C code

```
MQEINT32 value;
  rc = mqeFields_getInt32(hNewFields, &errStruct, &value, hFieldName);
```

MQe provides methods for storing and retrieving the following data types:
- A fixed length array is handled using the putArrayOf*type* and getArrayOf*type* methods, where *type* can be Byte, Short, Int, Long, Float, or Double.
- The ability to store variable length arrays is possible, but has been deprecated in this release. You can access these arrays using the Java put*type*Array and get*type***Array** calls or the C put*type* calls.
- The Java codebase has a slightly special form of operations for Float and Double types. This provides compatability with the MicroEdition. Floats are put using an Int representation and Doubles are put using a Long representation. Use the

`Float.floatToIntBits()` and `Double.doubleToLongBits()` to perform the conversion. However, this is not required on the C API.

Embedding MQeFields items

Description of how to embed an MQeFields item within another MQeFields item

An MQeFields item can be embedded within another MQeFields item by using the putFields and getFields methods.

The contents of an MQeFields item can be dumped in one of the following forms:

binary Binary form is normally used to send an MQeFields or MQeMsgObject object through the network. The dump method converts the data to binary. This method returns a binary byte array containing an encoded form of the contents of the item.

Note: This is not Java serialization.

When a fixed length array is dumped and the array does not contain any elements (its length is zero), its value is restored as null.

encoded string (Java only)
The string form uses the dumpToString method of the MQeFields item. It requires two parameters, a template and a title. The template is a pattern string showing how the MQeFields item data should be translated, as shown in the following example:

```
"(#0)#1=#2\r\n"
```

where

#0 is the data type (ascii or short, for example)

#1 is the field name

#2 is the string representation of the value

Any other characters are copied unchanged to the output string. The method successfully dumps embedded MQeFields objects to a string, but due to restrictions, the embedded MQeFields data may not be restored using the restoreFromString method.

Queues

Overview of MQe queues

What are MQe queues?

Introduction to MQe queues

MQe queues store messages. The queues are not directly visible to an application and all interactions with the queues take place through queue managers. For queue proxies, in the case of Java queue rules, refer the Rules topic. Each queue manager can have queues that it manages and owns. These queues are known as *local* queues. MQe also allows applications to access messages on queues that belong to another queue manager. These queues are known as *remote* queues. Similar sets of operations are available on both local and remote queues, with the exception of defining message listeners. Refer to “Message listeners” on page 77 for more information. The Queue types section provides more information on the different types of queue you can have.

Messages are held in the queue's persistent store. A queue accesses its persistent store through a *queue store adapter*. These adapters are interfaces between MQe and hardware devices, such as disks or networks, or software stores such as a database. Adapters are designed to be pluggable components, allowing the protocols available to talk to the device to be easily changed.

Queues may have characteristics, such as authentication, compression and encryption. These characteristics are used when a message object is stored on a queue. The *Security* topic provides more information on this.

Queue names

Constraints of MQe queue names

MQe queue names can contain the following characters:
- Numerics 0 to 9
- Lower case a to z
- Upper case A to Z
- Underscore _
- Period .
- Percent %

There are no inherent name length limitations in MQe.

Queues are configured using administration messages. Refer to the MQe Configuration Guide for more information on configuring MQe using administration messages.

Queue properties

Table of MQe queue properties

Queue properties are shown in the following table. Not all the properties shown apply to all the queue types:

Table 3. Queue properties

Property	Explanation	Java type	C type
Admin_Class	Queue class	String	admtype
Admin_Name	ASCII queue name	String	admname
Queue_Active	Queue in active/inactive state	boolean	qact
Queue_AttRule	Rule class controlling security operations	String	qar
Queue_Authenticator	Authenticator class	String	qau
Queue_BridgeName	Owning MQ bridge name	String	q-mq-bridge
Queue_ClientConnection	Client connection name	String	q-mq-client-con
Queue_CloseIdle	Close the connection to the remote queue manager once all messages have been transmitted	boolean	qcwi
Queue_CreationDate	Date that the queue was created	long	qcd
Queue_Compressor	Compressor class		qco
Queue_Cryptor	Cryptor class		qcr

Table 3. Queue properties (continued)

Property	Explanation	Java type	C type
Queue_CurrentSize	Number of messages on the queue		qcs
Queue_Description	Unicode description		qd
Queue_Expiry	Expiry time for messages		qe
Queue_ FileDesc	Location and adapter for the queue		qfd
Queue_MaxMsgSize	Maximum length of messages allowed on the queue		qms
Queue_MaxQSize	Maximum number of messages allowed		qmqs
Queue_Mode	Synchronous or asynchronous		qm
Queue_MQQMgr	MQ queue manager proxy		
Queue_Priority	Priority to be used for messages (unless overridden by a message value)		qp
Queue_QAliasNameList	Alternative names for the queue	String[]	qanl
Queue_QMgrName	Queue manager owning the real queue		qqmn
Queue_QMgrNameList	Queue manager targets		?
Queue_RemoteQName	Remote MQ field name		?
Queue_Rule	Rule class for queue operations		qr
Queue_QTimerInterval	Delay before processing pending messages		qti
Queue_TargetRegistry	Target registry type		qtr
Queue_Transporter	Transporter class		qtc
Queue_TransporterXOR	Transporter to use XOR compression		qtxor
Queue_Transformer	Transformer class		q-mq-transformer

Queue types

Introduction to MQe queue types

There are several different types of *queues* that you can use in an MQe environment.

Local queue

The simplest type of queue is a local queue. This type of queue is local to, and owned by, a specific queue manager. It is the final destination for all messages. Applications on the owning queue manager can interact directly with the queue to store messages in a safe and secure way, excluding hardware failures or loss of the device.

You can use local queues either online or offline, either connected or not connected to a network. Queues can also have security attributes set, in a very similar manner to protecting messages with attributes.

Access to messages on local queues is always synchronous, which means that the application waits until MQe returns after completing the operation, for example a put, get, or browse operation.

The queue owns access and security and may allow a remote queue manager to use these characteristics, when connected to a network. This allows others to send or receive messages to the queue.

Remote queue

A remote queue is a local queue belonging to another queue manager. This remote queue definition exchanges messages with the remote local queue.

MQe can establish remote queues automatically. If you attempt to access a queue on another queue manager, for example to send a message to that queue, MQe looks for a remote queue definition. If one exists it is used. If not, *queue discovery* occurs.

Note: The concept of queue discovery does not apply to the C codebase.

MQe discovers the authentication, cryptography, and compression characteristics of the real queue and creates a remote queue definition. Such queue discovery depends upon the target being accessible. If the target is not accessible, a remote definition must be supplied in some other way. When queue discovery occurs, MQe sets the access mode to synchronous, because the queue is now known to be synchronously available.

Synchronous remote queues are queues that can be accessed only when connected to a network that communicates with the owning queue manager. If the network is not established, the operations return an error. The owning queue controls the access permissions and security requirements needed to access the queue. It is the application's responsibility to handle any errors or retries when sending or receiving messages, because, in this case, MQe is no longer responsible for once and once-only assured delivery.

Asynchronous remote queues are queues used to send messages to remote queues and can store messages pending transmission. They cannot remotely retrieve messages. If the network connection is established, messages are sent to the owning queue manager and queue. However, if the network is not connected, messages are stored locally until there is a network connection and then the messages are transmitted. This allows applications to operate on the queue when the device is offline. As a result, these queues temporarily store messages at the sending queue manager while awaiting transmission.

Store-and-forward queue

Note: Store-and-forward queues are not implemented in the C codebase.

A store-and-forward queue stores messages on behalf of one or more remote queue managers until they are ready to receive them. This can be configured to perform either of the following:

- Push messages either to the target queue manager or to another queue manager between the sending and the target queue managers.
- Wait for the target queue manager to pull messages destined for it.

A store-and-forward queue stores messages associated with one or more target queue manager destinations. Messages addressed to a specific or target queue manager are placed on the relevant store-and-forward queue. The store-and-forward queue can optionally have a forwarding queue manager name set. If this name is set, the queue attempts to send all its messages to that named queue manager. If the name is not set, the queue just holds the messages.

Note: A store-and-forward queue and a *home server queue* should not have the same target queue manager. A store-and-forward queue with a queue QueueManagerName that is not the same as its host QueueManagerName, attempts to push messages to the remote queue manager. If that remote queue manager has a home server queue, it may attempt to pull the same message simultaneously, causing the message to lock.

Store-and-forward queues can hold messages for many target queue managers, or there may be one store-and-forward queue for each target queue manager.

This type of queue is normally, but not necessarily, defined on a server or gateway in Java only. Multiple store-and-forward queues can exist on a single queue manager, but the target names must not be duplicated. The contents of a store-and-forward queue are not available to application programs. Likewise a message sending application is quite unaware of the presence or role of store-and-forward queues in message transmission.

Dead-letter queue

MQe has a similar dead-letter queue concept to MQ. Such queues store messages that cannot be delivered. However, there are important differences in the manner in which they are used.

- In MQ, if a message is being moved from queue manager A to queue manager B, then if the target queue on queue manager B cannot be found, the message can be placed on the *receiving queue manager's* (B's) dead-letter queue.
- In MQe, if home-server queue on a client pulls a message from a server and is not able to deliver the message to a local queue and the client has a dead letter queue, the message will be placed on the client's dead letter queue.

  Note: In C, the Dead letter queue is just a local queue with a specific name.

  The use of dead-letter queues with an MQ bridge needs special consideration. For more information, see the topic on the MQ bridge.

Administration queue

The administration queue is a specialized queue that processes administration messages.

Messages put to the administration queue are processed internally. Because of this applications cannot get messages directly from the administration queue. Only one message is processed at a time, other messages that arrive while a message is being processed are queued up and processed in the sequence in which they arrive.

Home-server queue

This type of queue usually resides on a client and points to a store-and-forward queue on a server known as the *home-server*. The home-server queue pulls messages from the home-server store-and-forward queue when the client connects on the network.

In Java, home-server queues normally have a polling interval that causes them to check for any pending messages on the server while the network is connected.

When this queue pulls a message from the server, it uses assured message delivery to put the message to the local queue manager. The message is then stored on the target queue.

Home-server queues have an important role in enabling clients to receive messages over client-server connections.

MQ bridge queue

Note: The C codebase does not support MQ bridge queues.

This type of queue is always defined on an MQe gateway queue manager and provides a path from the MQe environment to the MQ environment. The MQ bridge queue is a remote queue definition that refers to a queue residing on an MQ queue manager.

Applications can use **put**, **get**, and **browse** operations on this type of queue, as if it were a local MQe queue.

Queue persistent storage

Overview of MQe message stores

Local queues and asynchronous remote queues store messages and therefore have properties to determine how and where the messages are stored.

The message store determines how the messages are mapped to the storage medium. The C and Java versions of MQe support a default message store, allowing long file names. The Java version of MQe has two additional message stores, `MQeShortFilenameMessageStore` that ensures the file name does not exceed eight characters, and the `MQe4690ShortFilenameMessageStore` that supports the default file system on a 4690. A storage adapter provides the message store access to the storage medium, the Java and C versions of MQe provide disk adapters with the Java version also providing a case insensitive adapter and a memory adapter.

The backing store used by a queue can be changed using an MQe administration message. Changing the backing store is not allowed while the queue is active or contains messages. If the backing store used by the queue allows the messages to be recovered in the event of a system failure, then this allows MQe to assure the delivery of messages.

MQe connection definitions

Explains how logical connections between queue managers are established

MQe supports a method of establishing logical connections between queue managers, in order to send or receive data.

MQe clients and servers communicate over connections called *client/server channels*.

Client/server channels have the following attributes:

- They are *dynamic*, that is created on demand. This differentiates them from MQ connections which have to be explicitly created.
- You can only establish the connection from the client-side.
- A client can connect to many servers, with each connection using a separate channel.
- The server-side queue manager can accept many connections simultaneously, from a multitude of different clients, using a listener for each protocol.
- They work through a Firewall, if the server-side of the connection is behind the Firewall. However, this depends on the configuration of the Firewall.
- They are *unidirectional* and support the full range of functions provided by MQe, including both synchronous and asynchronous messaging.

  Note: Unidirectional means that the client can send data to, or request data from the server, but the server-side cannot initiate requests of the client.

Standard connections, used for the client/server connection style, are unidirectional, but depend on a listener at the server, as servers cannot initiate data transfer. The client initiates the connection request and the server responds. A server can usually handle multiple incoming requests from clients. Over a standard connection, the client has access to resources on the server. If an application on the server needs synchronous access to resources on the client, a second connection is required where the roles are reversed. However, because standard connections are themselves bidirectional, messages destined for a client from its server's transmission queue, are delivered to it over the standard (client/server) connection that it initiated.

A client can be a client to multiple servers simultaneously. The client/server connection style is generally suited for use through Firewalls, because the target of the incoming connection is normally identified as being acceptable to the Firewall.

Note: Supposing there are two server queue managers, SQM1 and SQM2. SQM2 has listener address host 2: 8082. Also, suppose that SQM1 has a connection to SQM2 and a listener addresss, host 1:8081. If you create a connection definition on a client queue manager, named SQM2 with address host 1: 8081, this transports commands for SQM2 to SQM1, which then transports them to SQM2. Avoid this construct, as it is inefficient.

Because of the way channel security works, when a specific attribute rule is specified for a target queue, it forces the local queue manager to create an instance of the same attribute rule, `examples.rules.AttributeRule` and `com.ibm.mqe.MQeAttributeRule` are treated as the same rule. If this is not a desirable behaviour, you can specify a null rule for the target queue. In this case, com.ibm.mqe.`MQeAttributeDefaultRule` takes effect.

Connections can have various attributes or characteristics, such as authentication, cryptography, compression, or the transmission protocol to use. Different connections can use different characteristics. Each connection can have its own value set for each of the following attributes:

Authenticator
: This attribute causes authentication to be performed. This is a security function that challenges the putting application environment or user to prove their identity. It has a value of either `NULL` or an *authenticator* that can perform user or connection authentication.

Cryptor
: This attribute causes encryption and decryption to be performed on messages passing through the channel. This is a security function that encodes the messages during transit so that you cannot read them without the decoding information. Either null or a *cryptor* that can perform encryption and decryption.

  The simplest type of cryptor is MQeXorCryptor, which encrypts the data being sent by performing an exclusive-OR of the data. This encryption is not secure, but it modifies the data so that it cannot be viewed. In contrast, MQe3DESCryptor implements triple DES, a symmetric-key encryption method.

Channel
: The class providing the transport services.

Compressor
: This attribute causes compression and decompression to be performed on messages passing through the channel. This attempts to reduce the size of messages while they are being transmitted and stored. Either null or a *compressor* that can perform data compression and decompression. The simplest type of compressor is the MQeRleCompressor, which compresses the data by replacing repeated characters with a count.

Destination
: The server and port number for the connection. The target for this connection, for example SERVER.XYZ.COM

Typically, authentication only occurs when setting up the connection. All flows normally use compressors and cryptors.

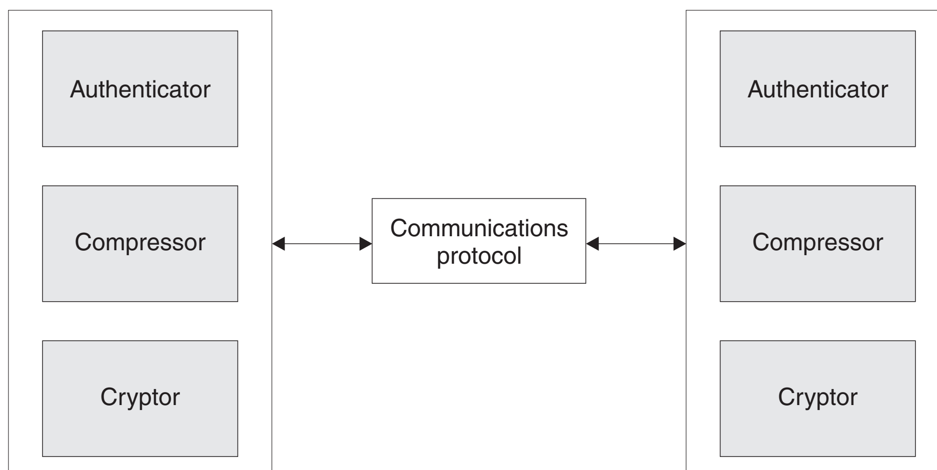


Figure 1. MQe connection

You can establish MQe connections using a variety of protocols allowing them to connect in a number of different ways, for example:

- Permanent connection, for example a LAN, or leased line

- Dial out connection, for example using a standard modem to connect to an Internet service provider (ISP)
- Dial out and answer connection, using a CellPhone, or ScreenPhone for example

MQe implements the communications protocols as a set of adapters, with one adapter for each of the supported protocols. This enables you to add new protocols.

Using queue aliases

Introduces the use of queue aliases

Aliases can be assigned for MQe queues to provide a level of indirection between the application and the real queues. Hence the attributes of a queue that an alias relates to can be changed without the application needing to change. For instance, a queue can be given a number of aliases and messages sent to any of these names will be accepted by the queue.

Examples of queue aliasing

Illustrates some of the ways in which aliasing can be used with queues

The following examples illustrate some of the ways in which aliasing can be used with queues:

Merging applications:

Using queue aliasing to merge applications

Suppose you have the following configuration:
- A client application that puts data to queue Q1
- A server application that takes data from Q1 for processing
- A client application that puts data to queue Q2
- A server application which takes data from Q2 for processing

Some time later the two server applications are merged into one application supporting requests from both the client applications. It may now be appropriate for the two queues to be changed to one queue. For example, you may delete Q2, and add an alias of the Q1 queue, calling it Q2. Messages from the client application that previously used Q2 are automatically sent to Q1.

Upgrading applications:

Using queue aliasing to upgrade applications

Suppose you have the following configuration:
- A queue Q1
- An application that gets messages from Q1
- An application that puts messages to Q1

You then develop a new version of the application that gets the messages. You can make the new application work with a queue called Q2. You can define a queue called Q2 and use it to exercise the new application. When you want it to go live, you let the old version clear all traffic off the Q1 queue, and then create an alias of Q2 called Q1. The application that puts to Q1 will still work, but the messages will end up on Q2.

Using different transfer modes to a single queue:

Using different transfer modes to a single queue, using queue aliasing

Suppose you have a queue MY_Q_ASYNC on queue manager MQE1. Messages are passed to MY_Q_ASYNC by a different queue manager MQE2, using a remote queue definition that is defined as an asynchronous queue. Now suppose your application periodically wants to get messages in a synchronous manner from the MY_Q_ASYNC queue.

The recommended way to achieve this is to add an alias to the MY_Q_ASYNC queue, perhaps called MY_Q_SYNC. Then define a remote queue definition on your MQE2 queue manager, that references the MY_Q_SYNC queue. This provides you with two remote queue definitions. If you use the MY_Q_ASYNC definition, the messages are transported asynchronously. If you use the MY_Q_SYNC definition, synchronous message transfer is used.

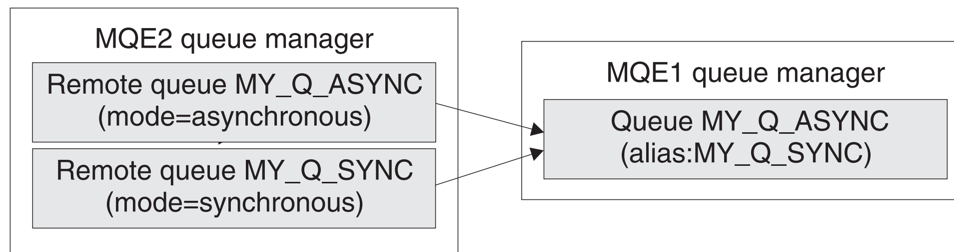


Figure 2. Two modes of transfer to a single queue

Queue manager operations

Explanation of the messaging operations that you can perform on a queue manager

This topic explains in detail the messaging operations that you can perform on a queue manager. It describes the services, functions, and uses of queue managers under the following headings:

What is an MQe queue manager

Introduction to the function and use of queue managers

The MQe queue manager is the focal point of the MQe system. It provides:

- A central point of access to a messaging and queueing network for MQe applications
- Optional client-side queuing
- Optional administration functions
- Once and once-only assured delivery of messages
- Recovery from failure conditions

- Extendable rules-based behavior

Unlike base MQ, MQe has a single queue manager type. However, you can program MQe queue managers to act as traditional clients or servers. You can also customize queue manager behavior using rules. The MQe queue manager is embedded within user written programs and these programs can run on any MQe supported device or platform.

You can configure queue managers in a number of different ways, the main types being client, server, and gateway. You can also update the queue store of a queue manager using administration messages. For more information on administration messages, refer to the MQe Configuration Guide.

an MQe queue manager can control the various types of queue. Communication with other queue managers on the MQ messaging network can be synchronous or asynchronous. If you want to use synchronous communications, the originator, and the target MQe queue managers must both be available on the network. Asynchronous communication allows an MQe application to send messages even when the remote queue manager is offline.

The queue manager life-cycle

Overview of the life-cycle of a queue manager

Typically, an application creates a new queue manager, configures it with a number of queues, and then frees the queue manager. An application also opens an existing queue manager, starts it, carries out messaging operations, and then stops. A further administration program can reopen the queue manager, remove all of its queues, and then stop. The following diagram displays this information:

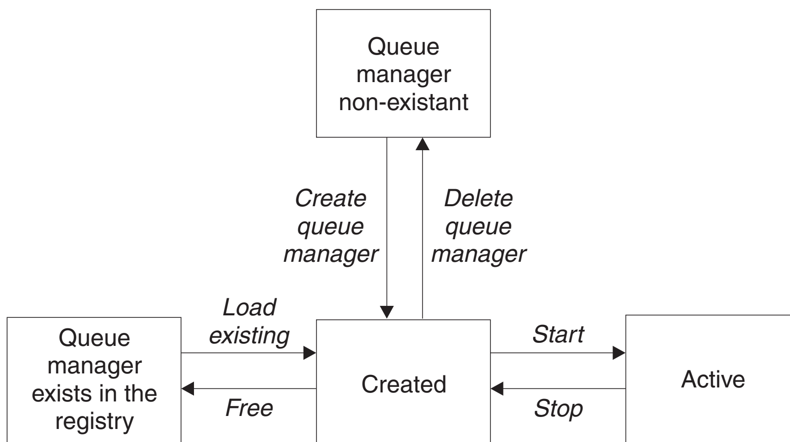


Figure 3. The queue manager life-cycle

Creating queue managers

A queue manager requires at least the following:
- A registry
- A queue manager definition
- Local default queue definitions

Once these definitions are in place you can run the queue manager and use the administration interface to perform further configuration, such as adding more queues.

Methods to create these initial objects are supplied in the MQeQueueManagerConfigure class.

The example install programs `examples.install.SimpleCreateQM` and `examples.install.SimpleDeleteQM` use this class.

Queue manager names

MQe queue manager names can contain the following characters:

- Numerics 0 to 9
- Lower case a to z
- Upper case A to Z
- Underscore _
- Period .
- Percent %

There are no inherent name length limitations in MQe.

Creating a queue manager - step by step

The basic steps required to create a queue manager are:

1. Create and activate an instance of MQeQueueManagerConfigure
2. Set queue manager properties and create the queue manager definition
3. Create definitions for the default queues
4. Close the MQeQueueManagerConfigure instance

Create and activate an instance of MQeQueueManagerConfigure:

You can activate the MQeQueueManagerConfigure class in either of the following ways:

1. Call the empty constructor followed by activate():

```
try
{
   MQeQueueManagerConfigure qmConfig;
   MQeFields parms = new MQeFields();
   // initialize the parameters

   qmConfig = new MQeQueueManagerConfigure( );
   qmConfig.activate( parms, "MsgLog:qmName\\Queues\\" );
}
catch (Exception e)
{ ... }
```

2. Call the constructor with parameters:

```
try
{
   MQeQueueManagerConfigure qmConfig;
   MQeFields parms = new MQeFields();
   // initialize the parameters
```

```
   qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
}
catch (Exception e)
{ ... }
```

The first parameter is an MQeFields object that contains initialization parameters for the queue manager. These must contain at least the following:

- An embedded MQeFields object (*Name*) that contains the name of the queue manager.
- An embedded MQeFields object, that contains the location of the local queue store as the registry type (*LocalRegType*) and the registry directory name (*DirName*). If a base file registry is used these are the only parameters that are required. If a private registry is used, a *PIN* and *KeyRingPassword* are also required.

The directory name is stored as part of the queue manager definition and is used as a default value for the queue store in any future queue definitions. The directory does not have to exist and will be created when needed.

If you use an alias for any of the initialization parameters, or if you wish to use an alias to set the connection attribute rule name, the aliases should be defined before activating MQeQueueManagerConfigure .

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
   MQeQueueManagerConfigure qmConfig;
   MQeFields parms = new MQeFields();
   // initialize the parameters
   MQeFields qmgrFields  = new MQeFields();
   MQeFields regFields   = new MQeFields();

   // Queue manager name is needed
   qmgrFields.putAscii(MQeQueueManager.Name, "qmName");
   // Registry information
   regFields.putAscii(MQeRegistry.LocalRegType,
                      "com.ibm.mqe.registry.MQeFileSession");
   regFields.putAscii(MQeRegistry.DirName, "qmname\\Registry");

   // add the imbedded MQeFields objects
   parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
   parms.putFields(MQeQueueManager.Registry, regFields);
   // activate the configure object
   qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
}
catch (Exception e)
{ ... }
```

Set queue manager properties:

When you have activated MQeQueueManagerConfigure, but before you create the queue manager definition, you can set some or all of the following queue manager properties:

- You can add a description to the queue manager with setDescription()
- You can set a connection time-out value with setChannelTimeout()
- You can set the name of the connection attribute rule with setChnlAttributeRuleName()

Call defineQueueManager( ) to create the queue manager definition. This creates a registry definition for the queue manager that includes any of the properties that you set previously.

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
   MQeQueueManagerConfigure qmConfig;
   MQeFields parms = new MQeFields();
   // initialize the parameters
   ...
   // activate the configure object
   qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
   qmConfig.setDescription("a test queue manager");
   qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
   qmConfig.defineQueueManager();
}
catch (Exception e)
{ ... }
```

At this point you can `close()` MQeQueueManagerConfigure and run the queue manager, however, it cannot do much because it has no queues. You cannot add queues using the administration interface, because the queue manager does not have an administration queue to service the administration messages.

The following sections show how to create queues and make the queue manager useful.

Create definitions for the default queues:

MQeQueueManagerConfigure allows you to define the following four standard queues for the queue manager:

defineDefaultAdminQueue()
: This administration queue is needed to allow the queue manager to respond to administration messages, for example to create new connection definitions and queues.

defineDefaultAdminReplyQueue()
: This administration reply queue is a local queue, used by connections as the destination of reply messages generated by administration.

defineDefaultDeadLetterQueue()
: This dead letter queue can be used, depending on the rules in force, to store messages that cannot be delivered to their correct destination.

defineDefaultSystemQueue()
: This default local queue, SYSTEM.DEFAULT.LOCAL.QUEUE, has no special significance within MQe itself, but it is useful when MQe is used with MQ messaging because it exists on every MQ messaging queue manager.

All methods throw an exception if the queue already exists.

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
   MQeQueueManagerConfigure qmConfig;
   MQeFields parms = new MQeFields();
   // initialize the parameters
```

```
   ...
   qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
   qmConfig.setDescription("a test queue manager");
   qmconfig.defineDefaultAdminQueue();
   qmconfig.defineDefaultAdminReplyQueue();
   qmconfig.defineDefaultDeadLetterQueue();
   qmconfig.defineDefaultSystemQueue();
}
catch (Exception e)
{ ... }
```

Close the MQeQueueManagerConfigure instance:

When you have defined the queue manager and the required queues, you can close() MQeQueueManagerConfigure and run the queue manager.

The complete example looks like this:

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
   MQeQueueManagerConfigure qmConfig;
   MQeFields parms = new MQeFields();
   // initialize the parameters
   MQeFields qmgrFields  = new MQeFields();
   MQeFields regFields   = new MQeFields();
   // Queue manager name is needed
   qmgrFields.putAscii(MQeQueueManager.Name, "qmName");
   // Registry information
   regFields.putAscii(MQeRegistry.LocalRegType, "com.ibm.mqe.registry.MQeFileSession");
   regFields.putAscii(MQeRegistry.DirName, "qmname\\Registry");
   // add the imbedded MQeFields objects
   parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
   parms.putFields(MQeQueueManager.Registry, regFields);
   // activate the configure object
   qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
   qmConfig.setDescription("a test queue manager");
   qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
   qmConfig.defineQueueManager();
   qmconfig.defineDefaultAdminQueue();
   qmconfig.defineDefaultAdminReplyQueue();
   qmconfig.defineDefaultDeadLetterQueue();
   qmconfig.defineDefaultSystemQueue();
   qmconfig.close();
}
catch (Exception e)
{ ... }
```

The registry definitions for the queue manager and the required queues are created immediately. The queues are not created until they are activated.

Persistent configuration data

MQe queue managers, irrespective of their role within the MQe network, require some information to be held in permanent storage. This is the responsibility of MQe. Iif there is additional information that must persist between invocations of an application, this is the responsibility of the application.

Information held within the registry contains Queue Manager configuration details, for example:

- Information on where messages, queues, remote queue definitions, channel timeout, aliases, adapters, and the message store are held and how to access them
- Connection definitions
- Security information
- Various bridge related objects

The following persistent information, useful to an application, is referred to in this manual as environmental data:
- Registry information, class, path, storage adapter class, and registry type. This information is used to locate an existing registry, allowing MQe to start an existing queue manager, or to create a new queue manager registry.
- Class manager information, for example class and name.
- Queue manager type.

Creating simple queue managers

The simplest MQe queue manager is a queue manager that uses a registry based upon the internal default values. The queue manager could be created without any queues, but its functionality would be severely limited. The example we create contains four standard queues:
- Admin queue - so that administration can be performed
- Admin reply queue - a standard place to store replies from administration actions
- System default queue - a useful general purpose local queue
- Dead letter queue - a place for undeliverable messages

The simplest queue manager has no security and has a registry stored in the local file system. The steps to achieve are:
- Create a registry on disk
- Create and start a queue manager using the registry
- Stop the queue manager

These actions are described for both the Java codebase and the C codebase, with example code for each. The example Java code is shipped as examples.config.CreateQueueManager. For C example code, refer to the HelloWorld compilation section and the transport-c file in the Broker example.

Creating a simple queue manager in Java:

Registries are created in Java by using the class com.ibm.mqe.MQeQueueManagerConfigure. An instance of this class is created, and activated by passing it some initialization parameters. The parameters are supplied in the form of an MQeFields object. Within this MQeFields are contained two sub fields, one holding information about the registry, and one holding information about the queue manager being created. As we are creating a very simple queue manager, we only need to pass two parameters, the queue manager name, in the queue manager parameters, and the registry location, in the registry parameters. We can then use the MQeQueue ManagerConfigure to create the standard queues.

First, create three fields objects, one for the QueueManager parameters, one for the Registry parameters. The third fields object, parms, is used to contain both the QueueManager and Registry fields objects.

```
MQeFields parms = new MQeFields();
MQeFields queueManagerParameters = new MQeFields();
MQeFields registryParameters = new MQeFields();
```

The QueueManager name needs to be set. Use the MQeQueueManager.Name as the Field Label constant.

```
queueManagerParameters.putAscii(MQeQueueManager.Name, queueManagerName);
```

The location of the persistent registry needs to be specified. Do this in the Registry Paramters field object. Use the MQeRegistry.DirName as the Field Label constant.

```
registryParameters.putAscii(MQeRegistry.DirName, registryLocation);
```

The QueueManager and registry parameters can now be set embedded the main fields object.

```
parms.putFields(MQeQueueManager.QueueManager, queueManagerParameters);
parms.putFields(MQeQueueManager.Registry, registryParameters);
```

An instance of MQeQueueManagerConfigure can be created now. This needs the parameters fields object, plus a String indentifying the details of the queue store to use.

```
MQeQueueManagerConfigure qmConfig =
new MQeQueueManagerConfigure(parms, queueStore);
```

The four common types of queues can now be created via four convenience methods as follows:

```
qmConfig.defineQueueManager();
qmConfig.defineDefaultSystemQueue();
qmConfig.defineDefaultDeadLetterQueue();
qmConfig.defineDefaultAdminReplyQueue();
qmConfig.defineDefaultAdminQueue();
```

Finally the MQeQueueManagerConfigure object can be closed.

```
qmConfig.close();
```

Creating a simple queue manager in C:

Stage 1: Create the admin components

All local administration actions can be accomplished using the MQeAdministrator. This allows you to create new QueueManagers and new Queues, and perform many other actions. For all calls, a pointer to the exception block is required, along with a pointer for the QueueManager handle.

Stage 2: Create a QueueManager

To create a QueueManager, two parameters structures are required. One contains the parameters for the QueueManager, the other for the registry. In this simple case the default values are suitable, with the addition of the location of the registry and queue store.

The call to the administrator will create the QueueManager. Note that the QueueManager name is passed into the call. A QueueManager Hndl is returned.

```
if ( MQERETURN_OK == rc ) {

   MQeQueueManagerParms qmParams  = QMGR_INIT_VAL;
```

```
        MQeRegistryParms     regParams = REGISTRY_INIT_VAL;

        qmParams.hQueueStore           = hQueueStore;
        qmParams.opFlags               = QMGR_Q_STORE_OP;
        regParams.hBaseLocationName    = hRegistryDir;

        display("Creating the Queue Manager\n");
        rc = mqeAdministrator_QueueManager_create(hAdministrator,
                                                  &exceptBlk,
                                                  &hQueueManager,
                                                  hLocalQMName,
                                                  &qmParams,
                                                  &regParams);

    }
```

Figure 4. Create queue manager C example

Starting queue managers

Queue managers need to be created before use. The creation step uses the QueueManagerConfigure Java class or the C administration API to create persistent queue manager data in a registry. The queue manager then uses the registry each time its starts.

Starting queue managers in Java

Normally, creating and starting a queue manager can require a large set of parameters. Therefore, the required parameters are supplied as an instance of MQeFields, storing the values as fields of correct type and name.

The parameters fall into two categories, queue manager parameters and registry parameters. Each of these categories is represented by its own MQeFields instance, and both are also enclosed in an MQeFields instance. The following Java example explains this concept, passing the queue managers name, ″ExampleQM″ and the location of a registry, ″C:\ExampleQM″:

```
/*create fields for queue manager parameters and place the queue manager name
MQeFields queueManagerParameters = new MQeFields();
queueManagerParameters.putAscii(MQeQueueManager.Name, "ExampleQM");

/*create fields for registry parameters and place the registry location
MQeFields registryParameters = new MQeFields();
registryParameters.putAscii(MQeRegistry.DirName, "C:\\ExampleQM\\registry");

/*create fields for combined parameters and place the two sub fields
MQeFields parameters = new MQeFields();
parameters.putFields(MQeQueueManager.Registry, queueManagerParameters);
parameters.putFields(MQeQueueManager.Registry, registryParameters);
```

Wherever you see ″`initialize the parameters`″ in code snippets, prepare a set of parameters as shown in the example, including the appropriate options. Only one queue manager name and one registry location are mandatory.

Starting a simple queue manager in Java:

To start the simplest queue manager, you only need to provide the queue manager name and registry location to the queue manager constructor. This starts and

activates the queue manager, and when the constructor returns the queue manager is running.

Figure 5. Start queue manager Java example

```
MQeQueueManager qm = newMQeQueueManager(queueManagerName, registryName);
```

There are other ways to start a queue manager that allow you to pass more parameters, in order to take advantage of some advanced features.

Starting queue managers in C

The `mqeQueueManager_new` function loads a queue manager for an established registry. To do this, you need information supplied by a queue manager parameter structure and a registry parameter structure.

The following example shows how you can set these structures to their default values, supplying only the directories of the queue store and registry:

```
MQeQueueManagerHndl hQueueManager;
MQeRegistryParms regParms = REGISTRY_INIT_VAL;
MQeQueueManagerParms qmParms = QMGR_INIT_VAL;
regParms.hBaseLocationName = hRegistryDirectory;
qmParms.hQueueStore = hStore;
qmParms.opFlags = QMGR_Q_STORE_OP;
rc = mqeQueueManager_new(&exceptBlock,
                         &hQueueManager, hQMName,
                         &regParams, &qmParms);
```

This creates a queue manager and loads its persistent information from the registry and creates queues. However, you must start the queue manager to:
- Create messages
- Get and put messages
- Process administration messages, using the administration queue

Note: In C, the queues are activated on starting the queue manager.

To start the queue manager, use

```
rc = mqeQueueManager_start(&hQueueManager, &exceptBlock);
```

Once the queue manager is started, messaging operations can take place and any queues that have messages on them are loaded.

To stop the queue manager, use:

```
rc = mqeQueueManager_stop(&hQueueManager, &exceptBlock);
```

Once stopped, you can restart the queue manager as required.

At the end of the application, you must free the queue manager to release any resources it uses, for example memory. First, stop the queue manager and then use:

```
rc = mqeQueueManager_free(&hQueueManager, &exceptBlock);
```

Starting a simple queue manager in C:

This process involves two steps:
1. Create the queue manager item.

2. Start the queue manager.

Creating the queue manager requires two sets of parameters, one set for the queue manager and one for the registry. Both sets of parameters are initialized. The *queue store* and the registry require directories.

Note: All calls require a pointer to ExceptBlock and a pointer to the queue manager handle.

```
 if (MQERETURN_OK == rc) {

   MQeQueueManagerParms qmParams  = QMGR_INIT_VAL;
   MQeRegistryParms     regParams = REGISTRY_INIT_VAL;
   qmParams.hQueueStore           = hQueueStore;
   qmParams.opFlags               = QMGR_Q_STORE_OP;

   /* ... create the registry parameters -
       minimum that are required */
   regParams.hBaseLocationName     =  hRegistryDir;
   display("Loading Queue Manager from registry \n");
   rc = mqeQueueManager_new(  &exceptBlock,
                            &hQueueManager,
                             hLocalQMName,
                             &qmParams,
                             &regParams);
}
```

You can now start the queue manager and carry out messaging operations:

```
 /* Start the queue manager  */

   if ( MQERETURN_OK == rc ) {
      display("Starting the Queue Manager\n");
      rc = mqeQueueManager_start(hQueueManager,
                        &exceptBlock);
   }
```

Queue manager parameters

List of the parameter names that can be passed to the queue manager and the registry.

The following lists the parameter names that you can pass to the queue manager and the registry:

Queue manager Parameters

MQeQueueManager.Name(ascii)
This is the name of the queue manager being started.

Registry Parameters

MQeRegistry.LocalRegType(ascii)
This is the type of registry being opened. MQe currently supports:

file registry
Set this parameter to `com.ibm.mqe.registry.MQeFileSession`.

private registry
Set this parameter to `com.ibm.mqe.registry.MQePrivateSession`.

You also need a private registry for some security features.

MQeRegistry.DirName(ascii)
This is the name of the directory holding the registry files. You must pass this parameter for a file registry.

MQeRegistry.PIN(ascii)
: You need this PIN for a private registry.

Note: For security reasons, MQe deletes the PIN and KeyRingPassword, if supplied, from the startup parameters as soon as the queue manager is activated.

MQeRegistry.CAIPAddrPort(ascii)
: You need this address and port number of a mini-certificate server for auto-registration, so that the queue manager can obtain its credentials from the mini-certificate server.

MQeRegistry.CertReqPIN(ascii)
: This is the certificate request number allocated by the mini-certificate administrator to allow the registry to obtain its credentials. You need this for auto-registration, so that the queue manager can obtain its credentials from the mini-certificate server.

MQeRegistry.Separator(ascii)
: This is used to specify a non-default separator. A separator is the character used between the the components of an entry name, for example `<QueueManager><Separator><Queue>`. Although this parameter is specified as a string, it must contain a single character. If it contains more than one, only the first character is used. Use the same separator for each registry opened and do not change it once a registry is in use. If you do not specify this parameter, the separator defaults to ″+″.

MQeRegistry.RegistryAdapter(ascii)
: This is the class, or an alias that resolves to a class, of the adapter that the registry uses to store its data. You must include this class if you want the registry to use an adapter other than the default MQeDiskFieldsAdapter. You can use any valid storage adapter class.

You always need the first two parameters. The last two are for auto-registration of the registry if it wishes to obtain credentials from the mini-certificate server.

MQeRegistry.RegistryAdapter (ascii)
: The class, (or an alias that resolves to a class), of the adapter that the registry uses to store its data. This value should be included if you want the registry to use an adapter other than the default MQeDiskFieldsAdapter. Any valid adapter class can be used.

A queue manager can run:
- As a client
- As server
- In a servlet

The following sections describe the example client, servers and servlet that are provided in the examples.queuemanager package. All queue managers are constructed from the same base MQe components, with some additions that give each its unique properties. MQe provides an example class, `MQeQueueManagerUtils`, that encapsulates many of the common functions.

All the examples require parameters at startup. These parameters are stored in standard ini files. The ini files are read and the data is converted into an MQeFields object. The `loadConfigFile()` method in the MQeQueueManagerUtils class performs this function.

Registry parameters for a queue manager

Description of the queue manager-related data held in the registry

The registry is the primary store for queue manager-related information; one exists for each queue manager. Every queue manager uses the registry to hold its:

- Queue manager configuration data
- Communications listener resource definitions
- Queue definitions
- Remote queue definitions
- Remote queue manager definitions
- User data, including configuration-dependent security information
- Optional bridge resource definitions

Registry type

MQE_REGISTRY_LOCAL_REG_TYPE
: The type of registry being opened. *file registry* and *private registry* are currently supported. A private registry is required for some of the security features.

For a file registry this parameter should be set to:

```
com.ibm.mqe.registry.MQeFileSession
```

For a private registry it should be set to:

```
com.ibm.mqe.registry.MQePrivateSession
```

Aliases can be used to represent these values.

Client queue managers

A client typically runs on a device platform, and provides a queue manager that can be used by applications on the device. It can open many connections to other queue managers.

A server usually runs for long periods of time, but clients are started and stopped on demand by the application that use them. If multiple applications want to share a client , the applications must coordinate the starting and stopping of the client.

Example - starting a client queue manager:

Starting a client queue manager involves:

1. Ensuring that there is no client already running. (Only one client is allowed per Java Virtual Machine.)
2. Adding any aliases to the system
3. Enabling trace if required
4. Starting the queue manager

The following code fragment starts a client queue manager:

```
/*-------------------------------------*/
/* Init - first stage setup            */
/*-------------------------------------*/
public void init( MQeFields parms ) throws Exception
{
  if ( queueManager != null )
/* One queue manager at a time   */
```

```
  {
    throw new Exception( "Client already running" );
  }
  sections = parms;
/* Remember startup parms        */
  MQeQueueManagerUtils.processAlias( sections );
/* set any alias names       */

// Uncomment the following line to start trace
    before the queue manager is started
//  MQeQueueManagerUtils.traceOn("MQeClient Trace", null);
/* Turn trace on   */

  /* Display the startup parameters */
  System.out.println( sections.dumpToString("#1\t=\t#2\r\n"));

  /* Start the queue manage  */
  queueManager = MQeQueueManagerUtils.processQueueManager( sections, null);
}
```

Once you have started the client, you can obtain a reference to the queue manager object either from the static class variable `MQeClient.queueManager` or by using the static method `MQeQueueManager.getReference(queueManagerName)`.

The following code fragment loads aliases into the system:

```
public static void processAlias( MQeFields sections ) throws Exception
{
  if ( sections.contains( Section_Alias ) )
/* section present ?            */
  {
/* ... yes                      */
    MQeFields section = sections.getFields( Section_Alias );
    Enumeration keys  = section.fields( );
/* get all the keywords         */
    while ( keys.hasMoreElements() )
/* as long as there are keywords*/
    {
      String    key   = (String) keys.nextElement();
/* get the Keyword   */
      MQe.alias( key, section.getAscii( key ).trim( ) );
/* add               */
    }
  }
}
```

Use the `processAlias` method to add each alias to the system. MQe and applications can use the aliases once they have been loaded.

Starting a queue manager involves:

1. Instantiating a queue manager. The name of the queue manager class to load is specified in the alias `QueueManager`. Use the MQe class loader to load the class and call the null constructor.
2. Activate the queue manager. Use the `activate` method, passing the MQeFields object representation of the ini file. The queue manager only makes use of the `[QueueManager]` and `[Registry]` sections from the startup parameters.

The following code fragment starts a queue manager:

```
public static MQeQueueManager processQueueManager( MQeFields sections,
     Hashtable ght ) throws Exception
{
/*                              */
  MQeQueueManager queueManager = null;
```

```
/* work variable                */
  if ( sections.contains( Section_QueueManager) )
/* section present ?    */
  {
/* ... yes                      */
   queueManager = (MQeQueueManager) MQe.loader.loadObject(Section_QueueManager);
    if ( queueManager != null )
/* is there a Q manager ?       */
    {
      queueManager.setGlobalHashTable( ght );
      queueManager.activate( sections );
/* ... yes, activate            */
    }
  }
  return( queueManager );
/* return the alloated mgr      */
}
```

Example - MQePrivateClient:

MQePrivateClient is an extension of MQeClient with the addition that it configures the queue manager and registry to allow for secure queues. For a secure client, the `[Registry]` section of the startup parameters is extended as follows:

```
(ascii)LocalRegType=PrivateRegistry

   Location of the registry

(ascii)DirName=.\ExampleQM\PrivateRegistry
   Adapter on which registry sits
(ascii)Adapter=RegistryAdapter
Network address of certificate authority

(ascii)CAIPAddrPort=9.20.7.219:8082
```

For MQePrivateClient and MQePrivateServer to work, the startup parameters must *not* contain `CertReqPIN`, *`KeyRingPassword`* and `CAIPAddrPort`.

Server queue managers

A server usually runs on a server platform. A server can run server-side applications but can also run client-side applications. As with clients, a server can open connections to many other queue managers on both servers and clients. One of the main characteristics that differentiate a server from a client is that it can handle many concurrent incoming requests. A server often acts as an entry point for many clients into an MQe network . MQe provides the following server examples:

MQeServer
: A console based server.

MQePrivateServer
: A console based server with enhanced security.

AwtMQeServer
: A graphical front end to MQeServer.

MQBridgeServer
: In addition to the normal MQe server functions, this server can send and receive messages to and from other members of the MQ family. This server is in package examples.mqbridge.queuemanager.

Example - MQeServer:

MQeServer is the simplest server implementation.

When two queue managers communicate with each other, MQe opens a connection between the two queue managers. The connection is a logical entity that is used as a queue manager to queue manager pipe. Multiple connections may be open at any time.

Server queue managers, unlike client queue managers, can have one or more listeners. A listener waits for communications from other queue managers, and processes incoming requests, usually by forwarding them to its owning queue manager. Each listener has a specified adapter that defines the protocol of incoming communications, and also specifies any extra data required.

You create listeners on the local queue manager using administration messages, remotely and locally. However, a remote queue manager must have a listener in order to receive a message.

A listener that has just been created by sending administration messages to the queue manager does not then start. To start it you can send an administration message explicitly to start the listener, or you can restart the queue manager. (However, listeners are persistent in the registry. This means that, once created, listeners that exist at queue manager startup are started automatically).

This example shows how to create and start a listener using administration messages:

```
String  listenerName = "MyListener";
     String  listenAdapter = "com.ibm.mqe.adapters.MQeTcpipHttpAdapter";
     int     listenPort = 1881;
     int     channelTimeout = 300000;
     int     maxChannels = 0;

        MQeCommunicationsListenerAdminMsg msg = new MQeCommunicationsListenerAdminMsg();

        msg.setName(listenerName);
        msg.create(listenAdapter, listenPort, channelTimeout, maxChannels);

        .
        .
        .

        //In order to start the listener use the start action

     MQeCommunicationsListenerAdminMsg msg = new MQeCommunicationsListenerAdminMsg();

     msg.setName(listenerName);
     msg.start();

     .
     .
```

When the listener is started, the server is ready to accept network requests.

When the server is deactivated:

1. The listener is stopped, preventing any new incoming requests
2. The queue manager is closed

Example - MQePrivateServer:

MQePrivateServer is an extension of MQeServer with the addition that it configures the queue manager and registry to allow for secure queues.

Environment relationship

This topic describes some requirements for running Java and C implementations of MQe.

Java code

The java queue manager runs inside an instance of a JVM. You can have only one queue manager per JVM. However, you can invoke multiple instances of the JVM.

Each of these queue managers must have a unique name. Java applications run inside the same JVM as the queue manager they use.

C code

You can run only one queue manager within a native C process. You need multiple processes for multiple queue managers. Each of these queue managers must have a unique name.

Stopping queue managers

Overview of stopping queue managers in Java and C

Stopping a queue manager in Java

There are 2 ways to close down a QueueManager, and one of the close methods should be called by MQe applications when they have finished using the queue manager:

- closeQuiese
- closeImmediate

closeQuiesce:

Stopping a queue manager using the closeQuiesce method

This method closes a Queue Manager, specifying a delay to allow existing internal processes to finish normally. Note that this delay is only implemented as a series of 100ms pause and retry cycles. Calling this method prevents any new activity, such as transmitting a message, from being started, but allows activities already in progress to complete. The delay is a suggestion only, and various JVM dependant thread scheduling factors could result in the delay being greater. If the activities currently in progress finish sooner, then the method returns before the expiry of the quiesce duration.

If the queue has not closed at the expiry of this period, it is forced to close.

After this method has been called, no more event notifications will be dispatched to message listeners. It is conceivable that messages may complete their arrival after this method has been called (and before it finishes). Such messages will not be notified. Application programmers should be aware of this, and not assume that every message arrival will generate a message event.

```
MQeQueueManager qmgr = new MQeQueueManager();
MQeMsgObject msgObj = null;
try {
```

```
  qmgr.putMessage(null, "MyQueue", msgObj, null, 0);
}  catch (MQeException e) {// Handle the exception here
}
qmgr.closeQuiesce(3000); // close QMgr
```

closeImmediate:

Stopping a queue manager using the closeImmediate method

This closes Queue Manager immediately.

After this method has been called, no more event notifications are dispatched to message listeners. Messages might complete their arrival after this method has been called, and before it finishes. Such messages are not notified, and therefore message arrival does not generate a message event.

```
MQeQueueManager qmgr = new MQeQueueManager();
MQeMsgObject msgObj = null;
try {
  qmgr.putMessage(null, "MyQueue", msgObj, null, 0);
} catch (MQeException e) {// Handle the exception here
}
qmgr.closeImmediate();  // close QMgr
```

Stopping a queue manager in C

Following the removal of the message from the queue, you can stop and free the queue manager. You can also free the strings that were created. Finally, terminate the session:

```
(void)mqeQueueManager_stop(hQueueManager,&exceptBlock);
(void)mqeQueueManager_free(hQueueManager,&exceptBlock);

/* Lets do some clean up */
(void)mqeString_free(hFieldLabel,&exceptBlock);
(void)mqeString_free(hLocalQMName,&exceptBlock);
(void)mqeString_free(hLocalQueueName,&exceptBlock);
(void)mqeString_free(hQueueStore,&exceptBlock);
(void)mqeString_free(hRegistryDir,&exceptBlock);


(void)mqeSession_terminate(&exceptBlock);
```

Deleting queue managers

This section details how to delete a queue manager in Java and C.

Java

Steps required to delete queue managers in Java

The basic steps required to delete a queue manager are:

1. Use the administration interface to delete any definitions
2. Create and activate an instance of `MQeQueueManagerConfigure`
3. Delete the standard queue and queue manager definitions
4. Close the `MQeQueueManagerConfigure` instance

When these steps are complete, the queue manager is deleted and can no longer be run. The queue definitions are deleted, but the queues themselves are not deleted. Any messages remaining on the queues are inaccessible.

Note: If there are messages on the queues they are not automatically deleted. Your application programs should include code to check for, and handle, remaining messages before deleting the queue manager.

1. Delete any definitions

You can use MQeQueueManagerConfigure to delete the standard queues that you created with it. Use the administration interface to delete any other queues before you call MQeQueueManagerConfigure.

2. Create and activate an instance of MQeQueueManagerConfigure

This process is the same as when creating a queue manager.

3. Delete the standard queue and queue manager definitions

Delete the default queues by calling:

- deleteAdminQueueDefinition() to delete the administration queue
- deleteAdminReplyQueueDefinition() to delete the administration reply queue
- deleteDeadLetterQueueDefinition() to delete the dead letter queue
- deleteSystemQueueDefinition() to delete the default local queue

These methods work successfully even if the queues do not exist.

Delete the queue manager definition by calling deleteQueueManagerDefinition()

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteAdminQueueDefinition();
 qmConfig.deleteAdminReplyQueueDefinition();
 qmConfig.deleteDeadLetterQueueDefinition();
 qmConfig.deleteSystemQueueDefinition();
 qmConfig.deleteQueueManagerDefinition();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

You can delete the default queue and queue manager definitions together by calling deleteStandardQMDefinitions(). This method is provided for convenience and is equivalent to:

```
deleteDeadLetterQueueDefinition();
deleteSystemQueueDefinition();
deleteAdminQueueDefinition();
deleteAdminReplyQueueDefinition();
deleteQueueManagerDefinition();
```

4. Close the MQeQueueManagerConfigure instance

When you have deleted the queue and queue manager definitions, you can close the MQeQueueManagerConfigure instance.

The complete example looks like this:

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
 qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteStandardQMDefinitions();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

C

Steps required to delete queue managers in C

The steps in deleting a queue manager are:

1. Remove all Connection Definitions.
2. Remove all Queues, including any ″system″ queues, for example the dead letter queue. Ensure all queues are empty.
3. Remove the queue manager.

You require an administrator to perform these functions. We also recommend stopping the queue manager first.

Note: Deleting the queue mananger will free the queue manager handle for you.

MQeAdministratorHndl hAdmin:

```
/* Create the new administrator based on the exisitng QM Handle */
rc = mqeAdministrator_new(&exceptBlock,
                  &hAdmin,hQueueManager);
if (MQERETURN_OK == rc) {

    if (MQERETURN_OK == rc) {
        /* delete any conncetion definitins for example :*/
        rc = mqeAdministrator_Connection_delete(hAdmin,
                                &exceptBlock,
                                hRemoteQM);
    }

    /* delete all the local queues here - remember to do "special*/
   /*queues" for example ... */
    if (MQERETURN_OK == rc) {
        rc = mqeAdministrator_LocalQueue_delete(hAdmin,
                                &exceptBlock,
                                MQE_DEADLETTER_QUEUE_NAME,
                                hLocalQMName);
    }

    /* Finally delete the queue manager */
    if (MQERETURN_OK == rc) {
```

```
        rc = mqeAdministrator_QueueManager_delete(hAdmin,
                                  &exceptBlock);
    }


    /* free of the amdinsitrator */
    (void)mqeAdministrator_free(hAdmin, &exceptBlock);
}
```

Messaging lifecycle

Description of the series of states through which a message progresses when it is put to a queue

When a message is put to a queue it progresses through a series of states. This section describes these states and related commands or events under the following headings:

Message states

Most queue types hold messages in a persistent store, for example a hard disk. While in the store, the state of the message varies as it is transferred into and out of the store. As shown in Figure 6:

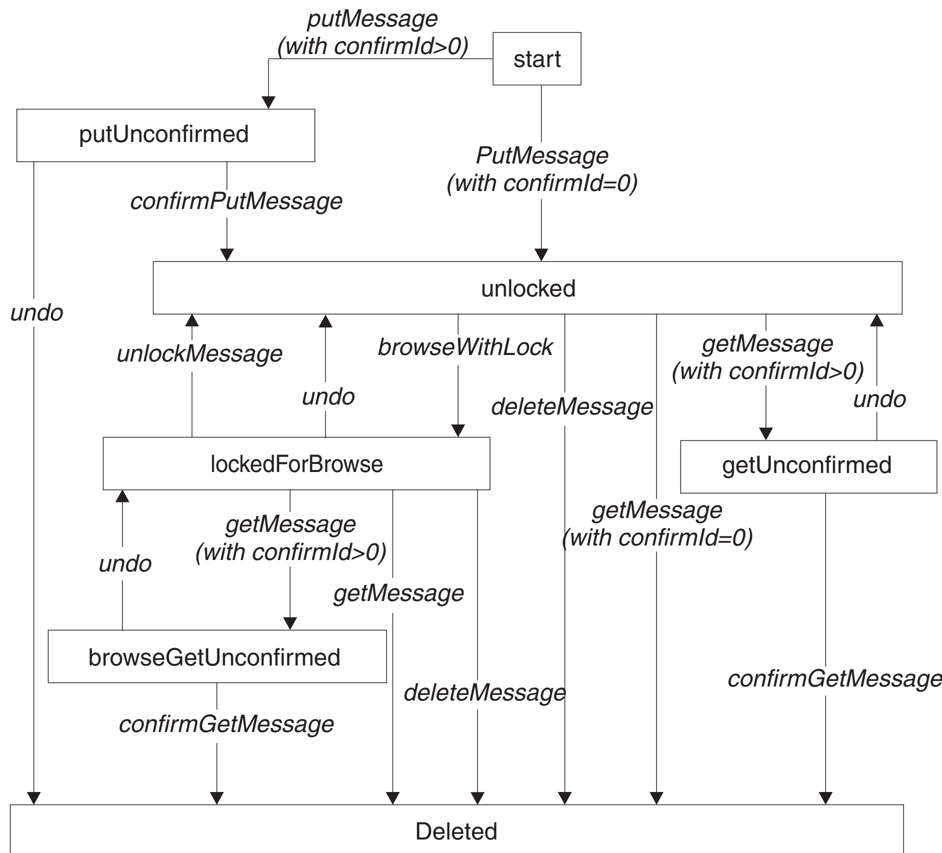


Figure 6. Stored message state flow

In this diagram, "start" and "deleted" are not actual message states. They are the entry and exit points of the state model. The message states are:

Put unConfirmed
: A message is put to the message store of a queue with a `confirmID`. The message is effectively hidden from all actions except `confirmPutMessage` or `undo`.

Unlocked
: A message has been put to a queue and is available to all operations.

Locked for Browse
: A browse with lock retrieves messages. Messages are hidden from all queries except `getMessage`, `unlock`, `delete`, `undo`, and `unlockMessage`. A `lockID` is returned from the browse operation. You must supply this `lockID` to all other operations.

Get Unconfirmed
: A `getMessage` call has been made with a `confirmID`, but the get has not been confirmed. The message is invisible to all queries except `confirmGetMessage`, `confirm`, and `undo`. Each of these actions requires the inclusion of the matching `confirmID` to confirm the get.

Browse Get Unconfirmed
: A message got while it is locked for browse. You can do this only by passing the correct `lockID` to the `getMessage` function.

On an asynchronous remote queue, other states exist where a message is being transmitted to another machine. These states are entered as "unlocked", that is only confirmed messages are transmitted.

Message events

Messages pass from one state to another as a result of an event. These events are typically generated by an API call. The possible message events, as shown in Figure 6 on page 69, are:

`putMessage`
: Places a message on a queue. This does not require a `confirmID`.

`getMessage`
: Retrieves a message from a queue. This does not require a `confirmID`.

`putMessage with confirmId>0`
: Places a message on a queue. This requires a `confirmID`. However, messages do not arrive at the receiving end in the order of sending, but in the order of confirmation.

`confirmPutMessage`
: A confirm for an earlier `putMessage` with a `confirmID`>0.

`getMessage with confirmId>0`
: Retrieves message from a queue. This requires a `confirmID`.

`confirmGetMessage`
: A confirm for an earlier `getMessage` with a `confirmID`>0.

`browseWithLock`
: Browses messages and lock those that match. Prevents messages from changing while browse is in operation.

unlockMessage
: Unlocks a message locked with a `browsewithLock` command.

undo
: Unlocks a message locked with a browse, undoes a `getMessage` with a `confirmID`>0, or undoes a putMessage with a `confirmID`>0.

deleteMessage
Removes a message from a queue.

Message index fields

Due to memory size constraints, complete messages are not held in memory, but, to enable faster message searching, MQe holds specific fields from each message in a *message index*. The fields that are held in the index are:

Java In Java, the following fields are held in the index:

UniqueID
`MQe.Msg_OriginQMgr + MQe.Msg_Time`

MessageID
`MQe.Msg_ID`

CorrelationID
`MQe.Msg_CorrelID`

Priority
`MQe.Msg_Priority`

C In C, the following fields are held in the index:

UniqueID
MQE_MSG_ORIGIN_QMGR + MQE_MSG_TIME

MessageID
MQE_MSG_MSGID

CorrelationID
MQE_MSG_CORRELID

Priority
MQE_MSG_PRIORITY

Providing these fields in a filter makes searching more efficient, since MQe may not have to load all the available messages into memory.

Messaging operations

The following table shows which types of messaging operations are valid on local queues, synchronous remote queues, and asynchronous remote queues. Note that the `Listen` and `Wait` operations are supported in Java only.

Table 4. Messaging operations on MQe queues

Operation	Local queue	Synchronous remote queue	Asynchronous remote queue
Put	Yes	Yes	Yes
Get	Yes	Yes	No
Browse	Yes	Yes	No
Delete	Yes	Yes	No
Listen	Yes	No	No
Wait	Yes	Yes	No

Note:

1. The synchronous remote wait operation is implemented through a poll of the remote queue, so the actual wait time is a multiple of the poll time
2. The MQ bridge supplied with MQe only supports an assured or unassured put, unassured get, and unassured browse (without lock).

Put

This operation places specified messages on a specified queue. The queue can belong to a local or remote queue manager. Puts to remote queues can occur immediately, or at a later time, depending on how the remote queue is defined on the local queue manager.

If a remote queue is defined as synchronous, message transmission occurs immediately. If a remote queue is defined as asynchronous, the message is stored within the local queue manager. The message remains there until it is transmitted. The put message call may finish before the message is put. Refer to "Message delivery" on page 82 for more information.

Note: In Java, if the local queue manager does not hold a definition of the remote queue then it attempts to contact the queue synchronously. This does not apply to the C codebase.

Assured delivery depends on the value of the `confirmID` parameter. Passing a non-zero value transmits the message as normal, but the message is locked on the target queue until a subsequent confirm is received. Passing a value of zero transmits the message without the need for a subsequent confirm. However, message delivery is not assured. Refer to "Message delivery" on page 82, for more information on assured and non-assured message delivery.

You can protect a message using message-level security.

Get

This operation returns an available message from a specified queue and removes the message from the queue. The queue can belong to a local or remote MQe queue manager, but cannot be an asynchronous remote queue.

If you do not specify a filter, the first available message is returned. If you do specify a filter, the first available message that matches the filter is returned. Including a valid `lockID` in the message filter allows you to get messages that have been locked by a previous browse operation. If no message is available, the get operation returns an error.

Using assured message delivery depends on the value of the `confirmID` parameter. Passing a non-zero value returns the message as normal. However, the message is locked and is not removed from the target queue until it receives a subsequent confirm. You can issue a confirm using the `confirmGetMessage()` method. However, message delivery is not assured. Refer to "Message delivery" on page 82, for more information on assured and non-assured message delivery.

Delete

This method deletes a message from a queue. It does not return the message to the application that called it. You must specify the UniqueID and you can delete only one message per operation.

The queue can belong to a local or synchronous remote MQe queue manager. Including a valid lockID in the message filter allows you to delete messages that have been locked by a previous operation, for example browse. If a message is not available, the application returns an error.

```
/* Example for deleting a message */
MQeFieldsHndl hMsg,hFilter;

/* create the new message */
rc = mqeFields_new(&exceptBlock, &hMsg);
if (MQERETURN_OK == rc) {

    /* add application fields here */
    /* ... */


    /* put message to a queue */
    rc = mqeQueueManager_putMessage(hQueueManager,
                        &exceptBlock,
                        hQMName,
                        hQueueName, hMsg,
                        NULL,0);
    if (MQERETURN_OK == rc) {
        /* Delete requires a filter -
        this can most easily be*/
      /*  found from the UID fields of the message*/
        rc = mqeFieldsHelper_getMsgUidFields(hMsg,
                              &exceptBlock,
                              &hFilter);
    }

}


/* some time later want to delete the message  -
    use the esatblished filter */
rc = mqeQueueManager_deleteMessage(hQueueManager,
                                 &exceptBlock,
                                  hQMName,
                                  hQueueName,
                                  hFilter);
```

Browse

You can browse queues for messages using a filter, for example message ID or priority . Browsing retrieves all the messages that match the filter, but leaves them on the queue. The queue can belong to a local or remote queue manager. However, the implementation of the browse command is codebase specific.

MQe also supports *Browsing under lock*. This allows you to lock the matching messages on the queue. You can lock messages individually, or in groups identified through a filter, and the locking operation returns a lockID. Use the lockID to get or delete messages. An option on browse allows you to return either the full messages, or only the UniqueIDs.

```
  MQeVectorHndl hListMsgs;

  rc = mqeQueueManager_browseMessages(hQueueManager,
                                    &exceptBlock,
                                    &hListMsgs,
                                    hQMName,
                                    hQueueName,
                                    hFilter,
                                   NULL,MQE_FALSE);
if (MQERETURN_OK == rc) {
```

```
    /* process list using mqeVector_* apis */

    /* free off the vector */
    rc = mqeVector_free(hListMsgs,&exceptBlock);
}
```

Returning an entire collection of messsages can be expensive in terms of system resources. Setting the `justUID` parameter to true and returns the `uniqueID` of each message that matches the filter only.

The messages returned in the collection are still visible to other MQe APIs. Therefore, when performing subsequent operations on the messages contained in the enumeration, the application must be aware that another application can process these messages once the collection is returned. To prevent other applications from processing messages, use the `browseMessagesAndLock` method to lock messages contained in the enumeration.

confirmPut

This method performs the confirmation of a previously successful `putMessage()` operation.

confirmGet

This method confirms the successful receipt of a message retrieved from a queue manager by a previous `getMessage()` operation. The message remains locked on the target queue until it receives a confirm flow.

Listen

Applications can listen for MQe message events, again with an optional filter. However, in order to do this, you must add a listener to a queue manager. Listeners are notified when messages arrive on a queue.

Wait

This method implements message polling. It allows you to specify a time for messages to arrive on a queue. Java implements a helper function for this. The C codebase, as it is non-threaded, must implement a function in application layer code. The following example demonstrates the `Wait` method:

Java Message polling uses the `waitForMessage()` method. This command issues a `getMessage()` command to the remote queue at regular intervals. As soon as a message that matches the supplied filter becomes available, it is returned to the calling application:

```
qmgr.waitForMessage("RemoteQMgr",
              "RemoteQueue",
              filter,
              null,
              0,
              60000);
```

The `waitForMessage()` method polls the remote queue for the length of time specified in its final parameter. The time is specified in milliseconds. Therefore, in the example, polling lasts for 6 seconds. This blocks the thread on which the command is running for 6 seconds, unless a message is returned earlier. Message polling works on both local and remote queues.

Note: Using this technique sends multiple requests over the network.

Queue ordering

Overview of the ordering of messages on a queue

The order of messages on a queue is primarily determined by their priority. Message priority ranges from 9 (highest) to 0 (lowest). Messages with the same priority value are ordered by the time at which they arrive on the queue, with messages that have been on the queue for the longest being at the head of the priority group.

Reading messages on a queue

If you issue a `getMessage` command when a queue is empty, the queue throws a Java codebase `Except_Q_NoMatchingMsg` exception or returns a C codebase `MQERETURN_QUEUE_ERROR`, `MQEREASON_NO_MATCHING_MSG`. This allows you to create an application that reads all the available messages on a queue.

Java

Encasing the `getMessage()` call inside a `try..catch` block allows you to test the code of the resulting exception. This is done using the `code()` method of the MQeException class. You can compare the result from the `code()` method with a list of exception constants published by the MQe class. If the exception is not of type `Except_Q_NoMatchingMsg`, throw the exception again.

The following code shows this technique:

```
try
{
  while(true)
    { /* keep getting messages until
      an exception is thrown    */
    MQeMsgObject msg = qmgr.getMessage( "myQMgr", "myQueue",
                            null, null, 0 );
    processMessage(msg);
    }
}
catch (Exception e)
{
    if ( e.code() != MQe.Except_Q_NoMatchingMsg )
    throw e;
}
```

Therefore, you can read all messages from a queue by iteratively getting messages until `MQe.Except_Q_NoMatchingMsg` is returned.

C

You can read all messages from a queue by looping, until the return code is `MQERETURN_QUEUE_WARNING` and the reason code is `MQEREASON_NO_MATCHING_MSG`.

Browse and Lock

Performing `BrowseAndLock` on a group of messages allows an application to ensure that no other application is able to process messages when they are locked. The messages remain locked until that application unlocks them. No other application can unlock the messages. Any messages that arrive on the queue after the `BrowseAndLock` operation are not locked.

An application can perform either a `get` or a `delete` operation on the messages to remove them from the queue. To do this, the application must supply the `lockID` that is returned with the enumeration of messages.

Specifying the `lockID` allows applications to work with locked messages without having to unlock them first.

Instead of removing the messages from the queue, it is also possible just to unlock them. This makes them visible once again to all MQe applications. You can achieve this by using the `unlockMessage` method.

Note: See the MQe Configuration Guide for special considerations with MQ bridge queues.

Example - Java:

Example of BrowseAndLock (Java)

The MQeEnumeration object contains all the messages that match the filter supplied to the browse. MQeEnumeration can be used in the same manner as the standard Java Enumeration. You can enumerate all the browsed messages as follows:

Note: You must supply a confirmID, in case the action of locating messages fails. It must be possible to undo the location, and this action requires the confirmID.

```
long confirmID = MQe.uniqueValue();
MQeEnumeration msgEnum = qmgr.browseMessagesAndLock( null,
                    "MyQueue",
                    null, null,
                             confirmID, false);

while( msgEnum.hasMoreElements() )
{
    MQeMsgObject msg = (MQeMsgObject)msgEnum.nextElement();
    System.out.println( "Message from  queue manager: " +
                        msg.getAscii( MQe.Msg_OriginQMgr ) );
}
```

The following code performs a `delete` on all the messages returned in the enumeration. The message's `UniqueID` and `lockID` are used as the filter on the `delete` operation:

```
while(msgEnum.hasMoreElements())
{
    MQeMsgObject msg = (MQeMsgObject)
                msgEnum.getNextMessage(null,0);

    processMessage(msg);

    MQeFields filter = msg.getMsgUIDFields();
    filter.putLong(MQe.Msg_LockID,
              msgEnum.getLockId());

    qmgr.deleteMessage(null, "MyQueue", filter);
}
```

Example - C:

Example of BrowseAndLock (C)

The C codebase example gets the actual message. Note the additional parameters, a confirmID in case the operation needs undoing, and the lockID.

```
MQeVectorHndl hMessages;
MQEINT64 lockID, confirmID=42;
rc = mqeQueueManager_browseAndLock(hQueueManager,
                            &exceptBlock,
                            &hmessages,
                            &lockID,
                            hQueueManagerName,
                            hQueueName,
                            hFilter,
                            NULL,      /*No Attribute*/
                            confirmID,
                            MQE_TRUE);    /*Just UIDs*/
/*process vector*/
MQeFieldsHndl hGetFilter;
rc = mqeFields_new(&exceptBlock, &hGetFilter);
if (MQERETURN_OK == rc){
  rc = mqeFields_putInt64(&hGetFilter,
                     &exceptBlock,
                     MQE_MSG_LOCKID,
                     lockID);
    if (MQERETURN_OK == rc){
      rc = mqeQueueManager_getMessage(&hQueueManager,
                             &exceptBlock,
                             hQueueManagerName,
                             hQueueName,
                            hGetFilter,
                            &hMsg);
}
```

Message listeners

Note: This section does not apply to the C codebase.

MQe allows an application to *listen* for events occurring on queues. The application is able to specify message filters to identify the messages in which it is interested, as shown in the following Java example:

```
/* Create a filter for "Order" messages of priority 7  */
MQeFields filter = new MQeFields();
filter.putAscii( "MsgType", "Order" );
filter.putByte( MQe.Msg_Priority, (byte)7 );
/* activate a listener on "MyQueue"        */
qmgr.addMessageListener( this, "MyQueue", filter );
```

The following parameters are passed to the addMessageListener() method:

- The name of the queue on which to listen for message operations
- A *callback* object that implements MQeMessageListenerInterface
- An MQeFields object containing a message filter

When a message arrives on a queue with a listener attached, the queue manager calls the callback object that it was given when the message listener was created.

The following is an example of the way in which an application would normally handle message events in Java:

```
public void messageArrived(MQeMessageEvent msgEvent)
  {
    String queueName =msgEvent.getQueueName();
    if (queueName.equals("MyQueue"))
    {
           try
```

```
        {
      /*get message from queue */
      MQeMsgObject msg =qmgr.getMessage(null,queueName,
            msgEvent.getMsgFields(),null,0);

      processMessage(msg );
           }
           catch (MQeException  e)
          {
           ...
          }
    }
  }
```

messageArrived() is a method implemented in MQeMessageListenerInterface. The `msgEvent` parameter contains information about the message, including:

- The name of the queue on which the message arrived
- The `UID` of the message
- The `messageID`
- The `correlationID`
- Message `priority`

Message filters only work on local queues. A separate technique known as polling allows messages to be obtained as soon as they arrive on remote queues.

Message polling

Note: This section does not apply to the C codebase.

Message polling uses the waitForMessage() method. This command issues a getMessage() command to the remote queue at regular intervals. As soon as a message that matches the supplied filter becomes available, it is returned to the calling application.

A wait for message call typically looks like this:

```
qmgr.waitForMessage( "RemoteQMgr", "RemoteQueue",
                filter, null, 0, 60000 );
```

The waitForMessage() method polls the remote queue for the length of time specified in its final parameter. The time is specified in milliseconds, so in the example above, the polling lasts for 60 seconds. The thread on which the command is executing is blocked for this length of time, unless a message is returned earlier.

Message polling works on both local and remote queues.

Note: Use of this technique results in multiple requests being sent over the network.

Trigger transmission

This method attempts to transmit pending messages. Only unlocked messages are transmitted.

Asynchronous remote queues and home server queues respond to trigger transmission processing. Put messages with no `confirmID` or put messages and confirm them before calling this method. Only messages that are fully ’put’ can be transmitted.

Trigger transmission rules

There are a number of rules, which can control the trigger transmission processing, if processing occurs. See the Rules topic for more information.

```
rc = mqeQueueManager_triggerTransmission(hQueueManager,&exceptBlock);
```

Servlet

Overview of servlet queue managers, which run inside a Web server

As well as running as a standalone server, a queue manager can be encapsulated in a servlet to run inside a Web server . A servlet queue manager has nearly the same capabilities as a server queue manager. MQeServlet provides an example implementation of a servlet. As with the server, servlets use ini files to hold start up parameters. A servlet uses many of the same MQe components as the server.

The main component not required in a servlet is the connection listener, this function is handled by the Web server itself. Web servers only handle http data streams so any MQe client that wishes to communicate with an MQe servlet must use the http adapter (com.ibm.mqe.adapters.MQeTcpipHttpAdaper). When you configure connections to queue managers running in servlets, you must specify the name of the servlet in the parameters field of the connection.

Example - configuring a connection on a servlet

The following definitions configure a connection on servlet /servlet/MQe with queue manager `PayrollQM`:

Connection name
: `PayrollQM`

Channel
: `com.ibm.mqe.communications.MQeChannel`

  Note: The com.ibm.mqe.MQeChannel class has been moved and is now known as com.ibm.mqe.communications.MQeChannel. Any references to the old class name in administration messages is replaced automatically with the new class name.

Channel Adapter
: `com.ibm.mqe.adapters.MQeTcpipAdapter:192.168.0.10:80`

Parameters
: `/servlet/MQe`

Options

Example - configuring a connection on a servlet using aliases

If the relevant aliases have been set up, you can configure the connection as follows:

Connection name
: `PayrollQM`

Channel
: `DefaultChannel`

Adapter
: `Network:192.168.0.10:80`

Parameters
/servlet/MQe

Options

Differences between server and servlet startup

The main differences compared to a server startup are:

- The servlet overrides the init method of the superclass. This method is called by the Web server to start the servlet. Typically this occurs when the first request for the servlet arrives.
- The name of the startup ini file cannot be passed in from the command line. The example expects to obtain the name using the servlet method getInitParameter() which takes the name of a parameter and returns a value. The MQe servlet uses a *Startup* parameter that it expects to contain an ini file name. The mechanism for configuring parameters in a Web server is Web server dependant.
- A listener is not started as the Web server handles all network requests on behalf of the servlet.
- As there is no listener a mechanism is required to time-out connections that have been inactive for longer than the time-out period. A simple timer class MQeChannelTimer is instantiated to perform this function. The *TimeInterval* value is the only parameter used from the `[Listener]` section of the ini file.

Example - starting a servlet

The MQe servlet extends javax.servlet.http.HttpServlet and overrides methods for starting, stopping and handling new requests. The following code fragment starts a servlet:

```
/**
 * Servlet initialization......
 */
public void init(ServletConfig sc) throws ServletException
{
  // Ensure supers constructor is called.
  super.init(sc);

  try
  {
    // Get the the server startup ini file
    String startupIni;
    if ((startupIni = getInitParameter("Startup")) == null)
      startupIni = defaultStartupInifile;

    // Load it
    MQeFields sections = MQeQueueManagerUtils.loadConfigFile(startupIni);

    // assign any class aliases
    MQeQueueManagerUtils.processAlias(sections);

    // Uncomment the following line to start trace before the queue
    // manager is started
    //      MQeQueueManagerUtils.traceOn("MQeServlet Trace", null);

    // Start connection manager
    channelManager = MQeQueueManagerUtils.processChannelManager(sections);

    // check for any pre-loaded classes
    loadTable = MQeQueueManagerUtils.processPreLoad(sections);

    // setup and activate the queue manager
    queueManager = MQeQueueManagerUtils.processQueueManager(sections,
```

```
     channelManager.getGlobalHashtable( ));

    // Start ChannelTimer  (convert time-out from secs to millisecs)
    int tI =
      sections.getFields(MQeQueueManagerUtils.Section_Listener).getInt
                                  ("TimeInterval");
    long timeInterval = 1000 * tI;
    channelTimer = new MQeChannelTimer(channelManager, timeInterval);

    // Servlet initialization complete
    mqe.trace(1300, null);
  }
  catch (Exception e)
  {
    mqe.trace(1301, e.toString());
    throw new ServletException(e.toString());
  }
}
```

Example - handling incoming requests

A servlet relies on the Web server for accepting and handling incoming requests. Once the Web server has decided that the request is for an MQe servlet, it passes the request to MQe using the doPost() method. The following code handles this request:

```
/**
 * Handle POST......
 */
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
              throws IOException
{
  // any request to process ?
  if (request == null)
    throw new IOException("Invalid request");
  try
  {
    int max_length_of_data = request.getContentLength();
    // data length
    byte[] httpInData = new byte[max_length_of_data];
    // allocate data area
    ServletOutputStream httpOut = response.getOutputStream();
    // output stream
    ServletInputStream  httpIn  = request.getInputStream();
    // input stream

    // get the request
    read( httpIn, httpInData, max_length_of_data);

    // process the request
    byte[] httpOutData = channelManager.process(null, httpInData);

    // appears to be an error in that content-
      length is not being set
    // so we will set it here
    response.setContentLength(httpOutData.length);
    response.setIntHeader("content-length", httpOutData.length);

    // Pass back the response
    httpOut.write(httpOutData);
  }
  catch (Exception e)
  {
```

```
      // pass it on ...
      throw new IOException( "Request failed" + e );
    }
}
```

This method:

1. Reads the http input data stream into a *byte array*. The input data stream may be buffered so the read() method is used to ensure that the entire data stream is read before continuing.

   Note: MQe only handles requests with the doPost() method, it does not accept requests using the doGet() method
2. The request is passed to MQe through a connection manager. From this point, all processing of the request is handled by core MQe classes such as the queue manager.
3. Once MQe has completed processing the request, it returns the result wrapped in http headers as a byte array. The byte array is passed to the Web server and is transmitted back to the client that originated the request.

Running multiple servlets on a web server

Web servers can run multiple servlets. It is possible to run multiple different MQe servlets within a Web server, with the following restrictions:

- Each servlet must have a unique name
- Only one queue manager is allowed per servlet
- Each MQe servlet must run in a different Java Virtual Machine (JVM)

Message delivery

Details of the different types of message delivery process

MQe networks are composed of connected queue managers and can include gateways. They can span multiple physical networks and route messages between them. In general they provide synchronous and asynchronous access to queues with a programming model that is independent of queue location.

Asynchronous message delivery

An asynchronous put to a remote queue places the message on the backing store associated with the local definition of that queue, along with its destination queue manager name, queue name, and the compressor, authenticator, and cryptor characteristics that match the target destination of the message. The message's dump method is called as it is saved to persistent storage in a secure format that is defined by its destination queue. The queue manager controls message delivery. It identifies or establishes a connection with appropriate characteristics to the queue manager for the next hop, then creates or reuses a transporter to the target queue manager. The transporter dumps the message and transmits the resulting byte string. The target queue manager and queue name are not part of that message flow.

If appropriate, the message is encrypted and compressed over the connection. If it has reached its destination queue manager, it is decrypted and decompressed. A new message is created, using the restore method, and the resultant message is placed on the destination queue. If the message has not reached its destination queue manager, it is decrypted and decompressed. It is then re-encrypted,

compressed, and placed on a store-and-forward queue for onward transmission, if a store-and-forward queue exists. In both cases it is held on its respective queue in a secure format, as defined by its destination queue.

A characteristic of asynchronous message delivery is that messages are passed to the queue manager at intermediate hops, being queued for onward transmission. Messages are taken off the intermediate queues first in order of priority, then in order of arrival on the queue. Duplicate messages, created when you resend a message, are also taken off the intermediate queues in the order of their arrival on the queue.

Synchronous message delivery

Synchronous message delivery is similar to the asynchronous case described above, but the queue manager involvement in intermediate hops takes place at a much lower level, involving the transporter and connections. An end-to-end connection is established, using the adapters defined in the protocol specifications at each intermediate node, to identify the next link. At the end of the last link, where no further relevant file descriptors exist, the message gets passed to the higher layers of the queue manager for processing. Thus the sending node does not queue the message but passes it along the connection, through intermediate hops, and then gives it to the destination queue manager to place it on the target queue.

The link into MQ uses a bridge queue on the gateway, which transforms the message into an MQ format. This mechanism means that synchronous MQe style messaging from a device is possible to MQ, with the connection terminating at the gateway. The message is delivered in real time from the gateway, through a client channel, to an MQ server. From there its destination can require it to be routed asynchronously along MQ message channels.

In a similar manner, a device capable of only synchronous messaging can send messages to an asynchronous MQe queue, provided that a suitable intermediary is available.

Assured and non-assured message delivery

Message delivery using synchronous message transmission can be assured or non-assured.

Assured message delivery

Asynchronous transmission introduces the concept of *assured message delivery*. When delivering messages asynchronously, MQe delivers each message once, and once-only, to its destination queue. However, this assurance is only valid if the definition of the remote queue and remote queue manager match the current characteristics of the remote queue and remote queue manager. If a remote queue definition and the remote queue do not match, then it is possible that a message may become undeliverable. In this case the message is not lost, but remains stored on the local queue manager.

Non-assured message delivery

Non-assured delivery of a message takes place in a single network flow. The queue manager sending the message creates or reuses a channel to the destination queue manager.

The message to be sent is dumped to create a byte-stream, and this byte stream is given to the channel for transmission. Once program control has returned from the channel the sender queue manager knows that the message has been successfully given to the target queue manager, that the target has logged the message on a queue, and that the message has been made visible to MQe applications.

However, a problem can occur if the sender receives an exception over the channel from the target. The sender has no way of knowing if the exception occurred before or after the message was logged and made visible. If the exception occurred before the message was made visible it is safe for the sender to send the message again. However, if the exception occurred after the message was made visible, there is a danger of introducing duplicate messages into the system since an MQe application could have processed the message before it was sent the second time.

The solution to this problem involves transmitting an additional confirmation flow. If the sender application receives a successful response to this flow, then it knows that the message has been delivered once and once-only.

Synchronous assured message delivery

You can perform assured message delivery using synchronous message transmission.

Put message - assured put

You can perform assured message delivery using synchronous message transmission, but the application must take responsibility for error handling.

The `confirmID` parameter of the `putMessage` method dictates whether a confirm flow is expected or not. A value of `zero` means that message transmission occurs in one flow, while a value of greater than zero means that a confirm flow is expected. The target queue manager logs the message to the destination queue as usual, but the message is locked and invisible to MQe applications, until a confirm flow is received. When you put messages with the `confirmID`, the messages are ordered by confirm time, not arrival time.

an MQe application can issue a `put` message confirmation using the `confirmPutMessage` method. Once the target queue manager receives the flow generated by this command, it unlocks the message, and makes it visible to MQe applications. You can confirm only one message at a time. It is not possible to confirm a batch of messages.

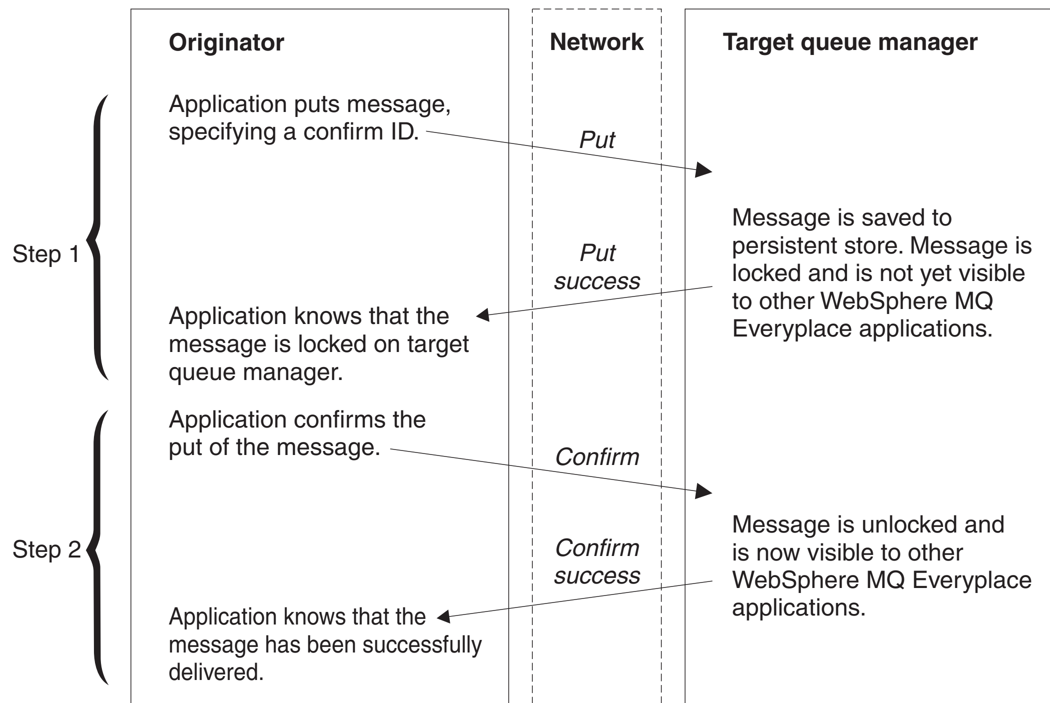


Figure 7. Assured put of synchronous messages

The `confirmPutMessage()` method requires you to specify the `UniqueID` of the message, not the `confirmID` used in the prior put message command. The `confirmID` is used to restore messages that remain locked after a transmission failure.

Example (Java) - assured put:

A skeleton version of the code required for an assured `put` is shown below:

```
long confirmId = MQe.uniqueValue();

try
{
  qmgr.putMessage( "RemoteQMgr", "RemoteQueue",
                   msg, null, confirmId );
}
catch( Exception e )
{
  /* handle any exceptions*/
}

try
{
    qmgr.confirmPutMessage( "RemoteQMgr", "RemoteQueue",
                            msg.getMsgUIDFields() );
}
catch ( Exception e )
{
  /* handle any exceptions    */
}
```

Example (C) - assured put:

A skeleton version of the code required for an assured `put` is shown below:

```
/* generate confirm Id */
MQEINT64 confirmId;
rc = mqe_uniqueValue(&exceptBlock,
                     &confirmId);

/* put message to queue using this confirm Id */
if(MQERETURN_OK == rc) {
    rc = mqeQueueManager_putMessage(hQMgr,
                         &exceptBlock,
                         hQMgrName, hQName,
                         hMsg, NULL, confirmId);
    /* now confirm the message put */
    if(MQERETURN_OK == rc) {
        /* first get the message uid fields */
           MQeFieldsHndl hFilter;
           rc = mqeFieldsHelper_getMsgUidFields(hMsg,
                                 &exceptBlock,
                                &hFilter);
          if(MQERETURN_OK == rc) {
            rc = mqeQueueManager_confirmPutMessage(hQMgr,
                                 &exceptBlock,
                                 hQMgrName,
                                 hQName, hFilter);
          }
    }
}
```

Exception handling - put message:

If a failure occurs during step 1 in "Put message - assured put" on page 84, the application should retransmit the message. There is no danger of introducing duplicate messages into the MQe network since the message at the target queue manager is not made visible to applications until the confirm flow has been successfully processed.

If the MQe application retransmits the message, it should also inform the target queue manager that this is happening. The target queue manager deletes any duplicate copy of the message that it already has. The application sets the `MQe.Msg_Resend` field to do this.

If a failure occurs during step 2 in "Put message - assured put" on page 84, the application should send the confirm flow again. There is no danger in doing this since the target queue manager ignores any confirm flows it receives for messages that it has already confirmed. This is shown in the following example, taken from the example program examples.application.example6.

Example - Java:

This example is taken from the examples.application.example6 example application:

```
boolean msgPut     = false;
  /* put successful?    */
boolean msgConfirm = false;
  /* confirm successful?    */
int maxRetry       = 5;
  /* maximum number of retries    */

long confirmId = MQe.uniqueValue();

int retry = 0;
while( !msgPut &&
        retry < maxRetry )
```

```
{
  try
  {
    qmgr.putMessage( "RemoteQMgr",
              "RemoteQueue",
              msg, null,
              confirmId );
   msgPut = true;
/* message put successful             */
  }
  catch( Exception e )
  {
    /* handle any exceptions  */
    /* set resend flag for
    retransmission of message  */
    msg.putBoolean( MQe.Msg_Resend, true );
    retry ++;
  }
}

if ( !msgPut )
  /* was put message successful?*/
    /* Number of retries has
    exceeded the maximum allowed,
   /*so abort the put*/
   /* message attempt */
return;

retry = 0;
while( !msgConfirm &&
          retry < maxRetry )
{
  try
  {
    qmgr.confirmPutMessage( "RenoteQMgr",
                "RemoteQueue",
                        msg.getMsgUIDFields());
    msgConfirm = true;
/* message confirm successful*/
  }
  catch ( Exception e )
  {
    /* handle any exceptions*/
    /* An Except_NotFound
    exception means */
  /*that the message has already    */
    /* been confirmed */
    if ( e instanceof MQeException &&
       ((MQeException)e).code() == Except_NotFound )
      putConfirmed = true;
    /* confirm successful */
    /* another type of exception -
    need to reconfirm message */
    retry ++;
  }
}
```

Example - C:

This example is taken from the examples.application.example6 example application:

```
MQEINT32 maxRetry = 5;

rc = mqeQueueManager_putMessage(hQMgr,
                    &exceptBlock,
                    hQMgrName,
```

```
                    hQName, hMsg,
                    NULL, confirmId);

/* if the put attempt fails,
    retry up to the maximum number*/
/*of retry times permitted,
    setting the re-send flag. */
while (MQERETURN_OK != rc
          && --maxRetry > 0 ) {
    rc = mqeFields_putBoolean(hMsg, &exceptBlock,
                      MQE_MSG_RESEND, MQE_TRUE);
    if(MQERETURN_OK == rc) {
       rc = mqeQueueManager_putMessage(hQMgr, &exceptBlock,
                             hQMgrName, hQName,
                             hMsg, NULL, confirmId);
    }
}

if(MQERETURN_OK == rc) {
    MQeFieldsHndl hFilter;
    maxRetry = 5;
    rc = mqeFieldsHelper_getMsgUidFields(hMsg,
                        &exceptBlock,
                        &hFilter);
    if(MQERETURN_OK == rc) {
          rc = mqeQueueManager_confirmPutMessage(hQMgr,
                           &exceptBlock,
                           hQMgrName, hQName,
                           hFilter);
    }
     while (MQERETURN_OK != rc
                && --maxRetry > 0 ) {
             rc = mqeQueueManager_confirmPutMessage(hQMgr,
                                   &exceptBlock,
                                     hQMgrName,
                                     hQName,
                                     hFilter);
    }
}
```

Get message - assured get

Assured message `get` works in a similar way to `put`. If a `get` message command is issued with a `confirmId` parameter greater than zero, the message is left locked on the queue on which it resides until a confirm flow is processed by the target queue manager. When a confirm flow is received, the message is deleted from the queue. Figure 8 on page 89 describes a get of synchronous messages:

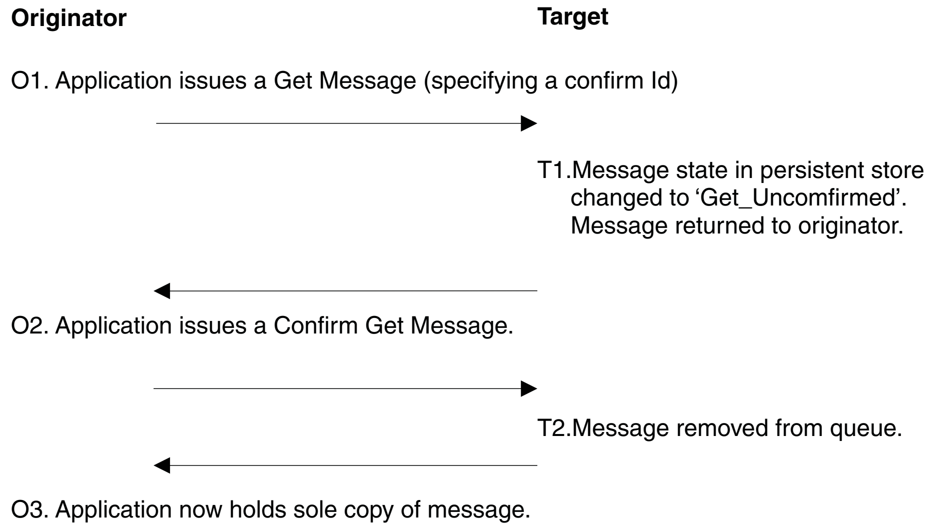


Figure 8. Assured get of synchronous messages

Example (Java) - assured get:

This example code is taken from the examples.application.example6 example program.

```
boolean msgGet     = false;
/* get successful?    */
boolean msgConfirm = false;
/* confirm successful?    */
MQeMsgObject msg   = null;
int maxRetry       = 5;
/* maximum number of retries    */

long confirmId = MQe.uniqueValue();
int retry = 0;
while( !msgGet && retry < maxRetry)
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr",
                           "RemoteQueue",
                           filter, null,
                           confirmId );
    msgGet = true;
  /* get succeeded    */
  }
  catch ( Exception e )
  {
    /* handle any exceptions */
    /* if the exception is of type
      Except_Q_NoMatchingMsg, meaning that   */
    /* the message is unavailable
      then throw the exception  */

    if ( e instanceof MQeException )
      if ( ((MQeException)e).code() ==
                    Except_Q_NoMatchingMsg )
        throw e;
    retry ++;
  /* increment retry count    */
  }
}

if ( !msgGet )
```

```
  /* was the get successful?       */
    /* Number of retry attempts has
    exceeded the maximum allowed, so abort  */
   /* get message operation   */
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr",
                            "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true;
  /* confirm succeeded    */
  }
  catch ( Exception e )
  {
    /* handle any exceptions */
    retry ++;    /* increment retry count */
  }
}
```

Example (C) - assured get:

This example code is taken from the examples.application.example6 example program.

```
MQEINT32 maxRetry = 5;

rc = mqeQueueManager_getMessage(hQMgr,
                           &exceptBlock,
                              hQMgrName,
                              hQName, hMsg,
                              NULL, confirmId);

/* if the get attempt fails, retry
    up to the maximum number of*/
/*retry times permitted,
  setting the re-send flag. */
while (MQERETURN_OK != rc  &&
                           --maxRetry > 0 ) {
    rc = mqeFields_getBoolean(hMsg,
                         &exceptBlock,
                            MQE_MSG_RESEND,
                            MQE_TRUE);
    if(MQERETURN_OK == rc) {
        rc = mqeQueueManager_getMessage(hQMgr,
                                &exceptBlock,
                                   hQMgrName,
                                   hQName, hMsg,
                                   NULL,
                                   confirmId);
    }
}

if(MQERETURN_OK == rc) {
    MQeFieldsHndl hFilter;
    maxRetry = 5;
    rc = mqeFieldsHelper_getMsgUidFields(hMsg,
                                   &exceptBlock,
                                   &hFilter);
    if(MQERETURN_OK == rc) {
          rc = mqeQueueManager_confirmGetMessage(hQMgr,
                                        &exceptBlock,
                                           hQMgrName,
                                           hQName,
```

```
                                          hFilter);
    }
     while (MQERETURN_OK != rc  &&
                          --maxRetry > 0 ) {
             rc = mqeQueueManager_confirmPutMessage(hQMgr,
                                             &exceptBlock,
                                                hQMgrName,
                                                hQName,
                                                hFilter);
    }
}
```

Undo command:

The value passed as the confirmId parameter also has another use. The value is used to identify the message while it is locked and awaiting confirmation. If an error occurs during a get operation, it can potentially leave the message locked on the queue. This happens if the message is locked in response to the get command, but an error occurs before the application receives the message. If the application reissues the get in response to the exception, then it will be unable to obtain the same message because it is locked and invisible to MQe applications.

However, the application that issued the get command can restore the messages using the undo method. The application must supply the confirmId value that it supplied to the get message command. The undo command restores messages to the state they were in before the get command.

The undo command also has relevance for the putMessage and browseMessagesAndLock commands. As with get message, the undo command restores any messages locked by the browseMessagesandLock command to their previous state.

If an application issues an undo command after a failed putMessage command, then any message locked on the target queue awaiting confirmation is deleted.

The undo command works for operations on both local and remote queues.

Undo command example - Java:

```
boolean msgGet     = false;
/* get successful?    */
boolean msgConfirm = false;
/* confirm successful?    */
MQeMsgObject msg   = null;
int maxRetry       = 5;
/* maximum number of retries    */

long confirmId = MQe.uniqueValue();
int retry = 0;
while( !msgGet && retry < maxRetry )
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr",
                    "RemoteQueue",
                    filter, null,
                            confirmId );
    msgGet = true;
  /* get succeeded    */
  }
  catch ( Exception e )
  {
    /* handle any exceptions   */
```

```
    /* if the exception is of type
      Except_Q_NoMatchingMsg, meaning that   */
    /* the message is unavailable
      then throw the exception */
    if ( e instanceof MQeException )
      if ( ((MQeException)e).code() == Except_Q_NoMatchingMsg )
        throw e;
    retry ++;    /* increment retry count    */
    /* As a precaution, undo the message
      on the queue. This will remove   */
    /* any lock that may have been put on
      the message prior to the        */
    /* exception occurring   */
    myQM.undo( qMgrName, queueName, confirmId );
  }
}

if ( !msgGet )
    /* was the get successful?        */
      /* Number of retry attempts has
      exceeded the maximum allowed, so abort  */
      /* get message operation   */
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr",
                    "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true;
  /* confirm succeeded    */
  }
  catch ( Exception e )
  {
    /* handle any exceptions         */
    retry ++;
  /* increment retry count    */
  }
}
```

Undo command example - C:

```
MQeFieldsHndl hMsg;
rc = mqeQueueManager_getMessage(hQMgr, &exceptBlock,
                      &hMsg, hQMgrName,
                        hQName, hFilter,
                        NULL, confirmId);
/* if unsuccessful, undo the operation */
if(MQERETURN_OK != rc) {
    rc = mqeQueueManager_undo(hQMgr, &exceptBlock,
                      hQMgrName, hQName,
                      confirmId);
}
```

Network topologies and message resolution

Introduction to message routes and their use with MQe

Overview

This topic explains, in detail, the concept of message routes and how to use them with MQe.

Several features of MQe allow the routing of messages to be altered dynamically. However, you need to ensure that there are no 'in doubt' messages that would be affected by the change. If a message is put with a non-zero confirm ID, and then the MQe network topology is changed to alter the routing of the subsequent confirmGetMessage call, the unconfirmed message will not be found. MQe protocol treats a failure to confirm a put as an indication that the put message has been confirmed already, and therefore assumes success. This could leave an unconfirmed message on a queue, which represents a loss of a message, and therefore breaks the assured delivery promise.

Since MQe uses the same two step process to assure delivery of asynchronously sent messages, regardless of whether a zero or non-zero confirmId is used, changing the network topology can break the assured delivery of asynchronous message sends.

Notation

The topics within *Network topologies and message resolution* use a consistent notation for illustrating the resources. This allows the areas of specific interest to be shown prominently, while the less relevant parts of a system can be hidden. This is easier to show with a diagram:

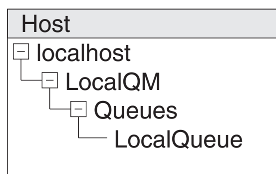


Figure 9. A host and the MQe resources on it

The following diagram shows the same resources in the 'dispersed' form:

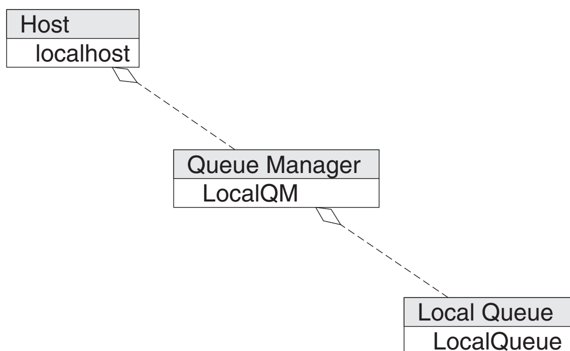


Figure 10. A host and the MQe resources on it: 'dispersed' form

The line with a diamond shape shows that the queue manager is the child of the host. This preserves the parent/child relationship from the tree, which would otherwise be lost by separating the elements.

Introduction

The route that a message takes through an MQe network can depend upon many resources (queues, connection definitions, listeners and so on). These need to be correctly set up, often in pairs whose settings need to be complementary. Failure to

set up the correct resources, or setting certain of their values incorrectly can result in failure to deliver messages. Since the task of setting up a network that correctly routes messages can initially appear complex, this topic describes the theory underlying message resolution.

A common source of confusion with MQe is the differentiation between a local queue that exists on a remote machine (or queue manager), and a local definition of that queue on the remote machine. Both of these entities are commonly referred to as 'remote queue's. In order to clarify these, the term 'remote queue reference' is used to describe a local definition of a queue that resides on another (remote) machine (or queue manager).

Local queue resolution

Local message putting is fundamental to MQe. Messages, if they are to be useful, must always end up on a local queue. Message route resolution is the mechanism by which a message travels through an MQe network to its ultimate destination.

The following diagram shows a simple local message put.

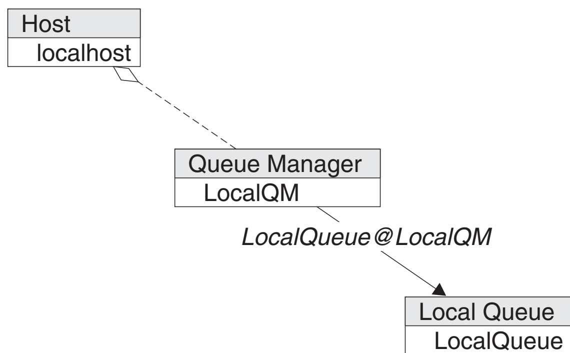


Figure 11. A simple local message put

The message route is shown for a message put to (QueueManager)LocalQM destined for the (Queue)LocalQueue@LocalQM. This is clearly a put to a local queue, as the queue's 'queue manager name' is the same as the name of the queue manager to which the message is put.

The message route is shown with an arrow labelled with the message route name. The arrow indicates the direction in which the message flows. The text on the label indicates the currently used target name (this can change during message resolution). LocalQM looks for a queue to accept a message for LocalQueue@LocalQM. The process of determining which queue to place a message on is called Queue Resolution. LocalQM finds an exact match for the destination, the local queue. It then puts the message onto the local queue. The message will then reside on the local queue until it is retrieved via the getMessage() API call.

Local queue alias

Local queues can have aliases. If we add a queue alias to the local queue we provide it with another name by which it will be known. So the local queue LocalQueue@LocalQM could be given an alias of 'LocalQueueAlias', as shown in the following diagram:

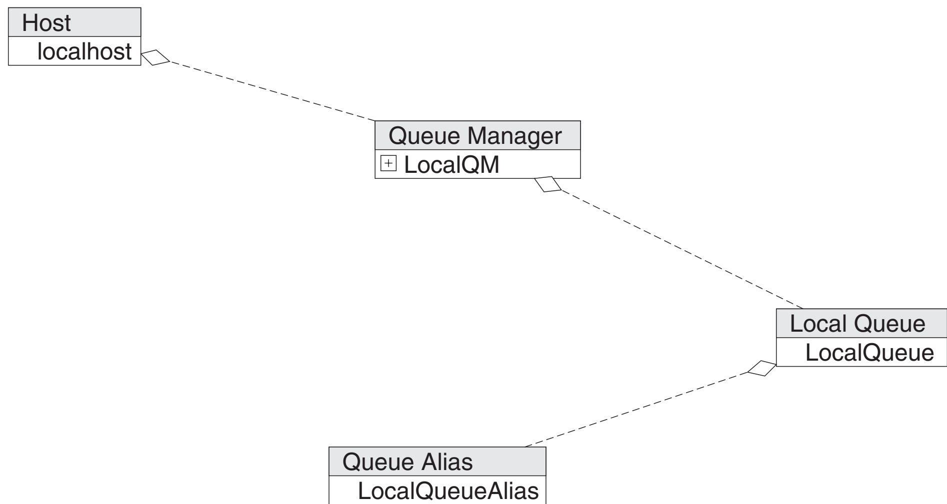


Figure 12. LocalQueue@LocalQM with an alias of 'QueueAlias'.

Messages addressed to LocalQueueAlias@LocalQM would be directed by the queue manager to LocalQueue@LocalQM. We could envisage this as the message being placed on the matching alias, almost as if the alias were a queue, and then the alias moves the message to the correct destination, as shown in the following diagram:

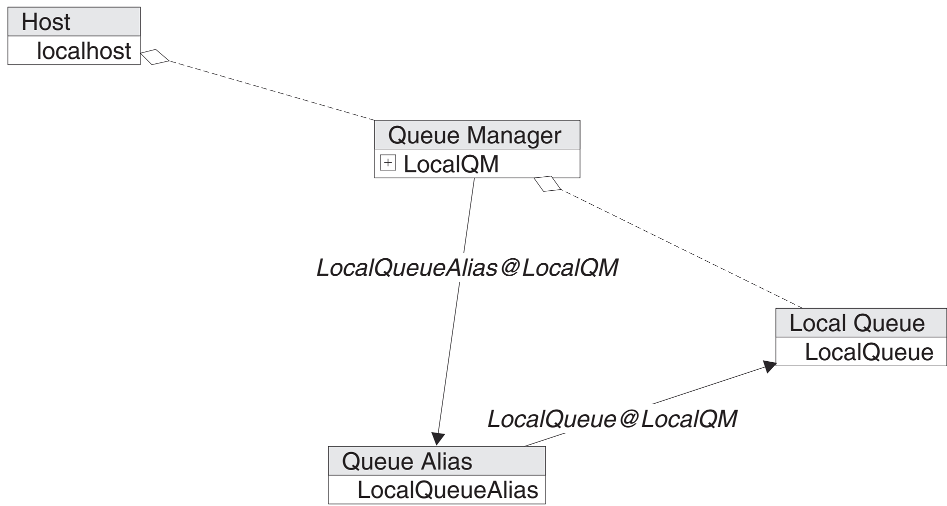


Figure 13. A message being placed on a matching alias

The redirection of the message by the alias is accompanied by a change in the 'destination queue name' from LocalQueueAlias@LocalQM to LocalQueue@LocalQM. The fact that the message was originally put to the alias is completely lost. This can be seen by the labelling of the message route from the alias to the queue. In this particular case the change of 'put name' is of little or no importance, but this is important in more complex message resolutions.

The resolution of the queue alias is performed just before the message is routed to the queue. The resolution is as late as it could possibly be, and is sometimes termed 'late resolution'.

Queue manager alias

Queue aliases enable you to refer to queues by more than one name. Queue Manager Aliases enable you to refer to queue managers by more than one name. We can define a Queue Manager Alias 'AliasQM' referring to the local queue manager, as shown in the following diagram:

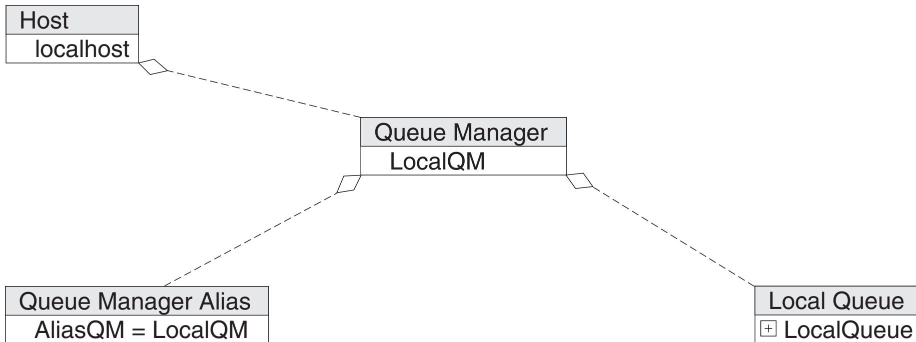


Figure 14. Defining a queue manager alias

Messages addressed to 'AliasQM' are routed to 'LocalQM', as shown in the following diagram:

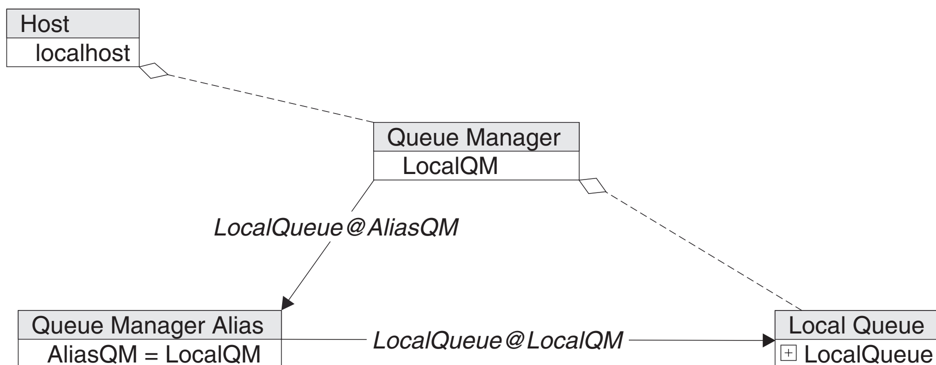


Figure 15. Addressing messages to a queue manager alias

The redirection of the message by the alias is accompanied by a change in the 'destination queue name' from LocalQueue@AliasQM to LocalQueue@LocalQM. The fact that the message was originally put to the alias is completely lost. This can be seen by the labelling of the message route from the alias to the queue. Queue Manager Aliases are resolved at the beginning of message resolution. Queue Manager Aliases are very effective as part of complex topologies

To complete the picture we can resolve both the Queue Manager Alias and the Queue Alias, as shown in the following diagram:

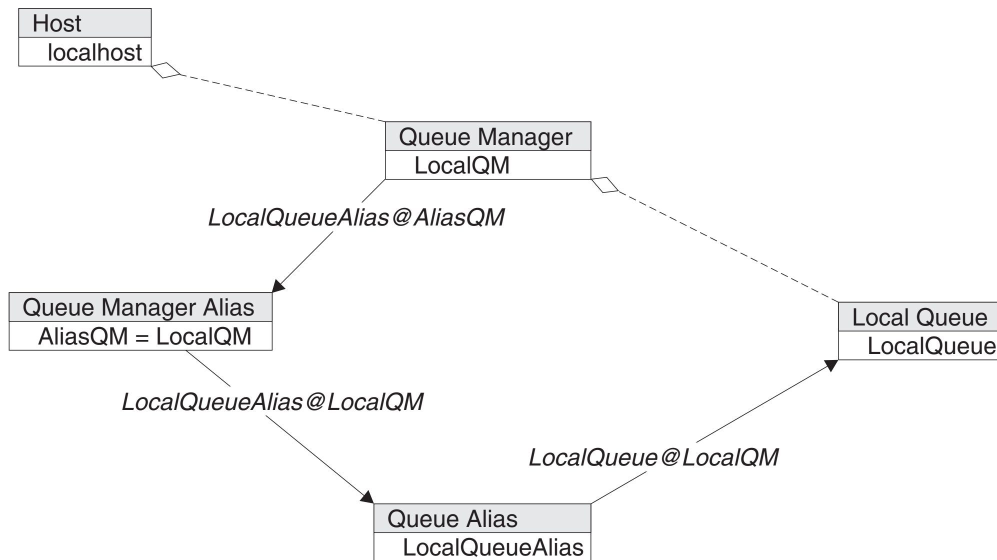


Figure 16. Resolving the queue manager alias and the queue alias

Here we put a message to LocalQueueAlias@AliasQM, and it is resolved first via the Queue Manager Alias, and then through the Queue Alias.

Resolution of queueManager aliases happens as soon as the request reaches a queue manager. The effect is to substitute the aliased string for the aliasing string. So for the first example above, as soon as the putMessage(″AliasQM″,....) call crosses the API, it is converted to a putMessage(″LocalQM″,....) call. This resolution is also performed when a message is put to a remote queue manager. On a remote queue manager the queue aliases on that queue manager are used, not those on the originating queue manager.

An alias can point to another alias. However, circular definitions have unpredictable results. An alias can also be made of the local queue manager name. This allows a queue manager to behave as if it were another queue manager. This pretence means that we can remove a queue manager entirely from the network, and by creating suitable queue manager aliases elsewhere we can allocate its workload to another queue manager. This feature is useful when modifying MQe network topologies, because servers, under the control of system administrators, can be moved, removed or renamed without breaking the connectivity of clients, which may not be so readily accessible.

Remote queue resolution

Remote queue resolution involves connection definitions and network resolution. It requires a setup where there are two queue managers, one of which is the local queue manager that you use to put the message, and the other is the queue manager to which you want the message to go. The remote queue manager must have a listener, and the local queue manager must have a connection definition describing the listener, as shown in the following diagram:

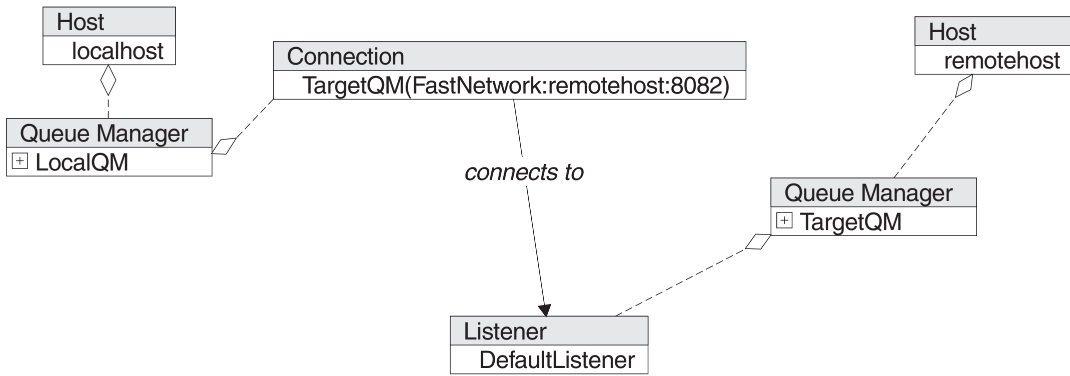


Figure 17. Local and remote queue managers with a definition and listener pair

The connection definition/listener pair allows MQe to establish the network communications necessary to flow the message. The connection definition contains information about communicating with a single queue manager. The connection definition is named for the queue manager to which it defines a route. So in this example the connection definition is called TargetQM, and contains the information necessary to establish connection with (QueueManager)TargetQM. This information includes the address of the machine upon which the queue manager resides (remote host in this example), the port upon which the queue manager is listening (8081 in this example), and the protocol to use when conversing with the queue manager (FastNetwork in this example).

You need a remote queue reference on LocalQM representing the destination queue TargetQueue which resides on TargetQM. There are therefore two entities called TargetQueue@TargetQM. One is the 'real' queue, that is a local queue, and one is a reference to the real queue, a remote queue reference, as shown in the following diagram:

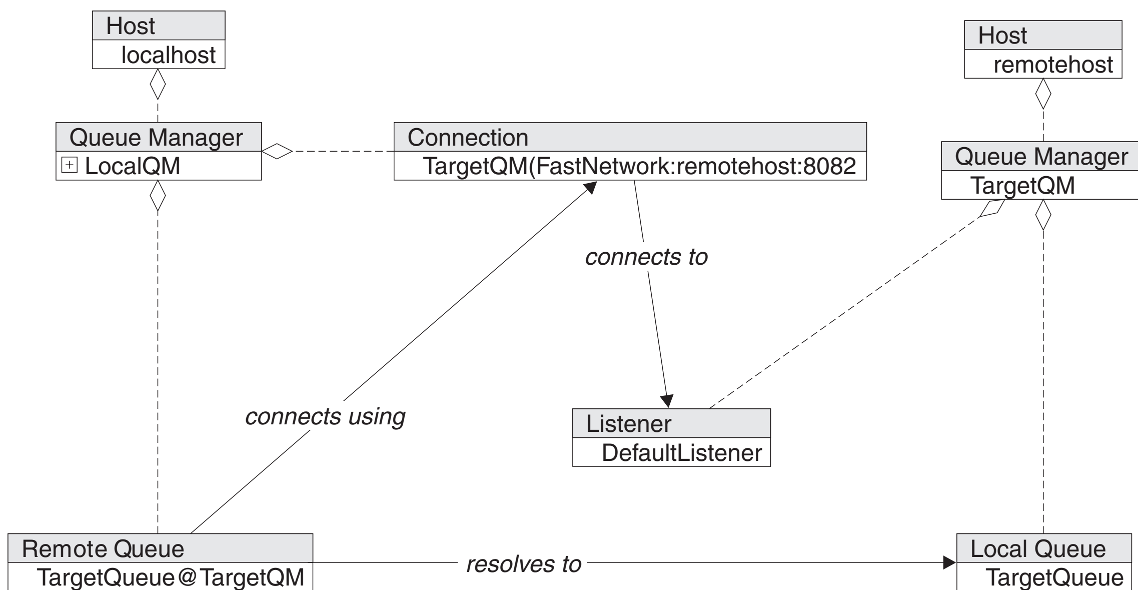


Figure 18. A remote queue reference.

The message resolution for a put on LocalQM to TargetQueue@TargetQM works as shown in the following diagram:

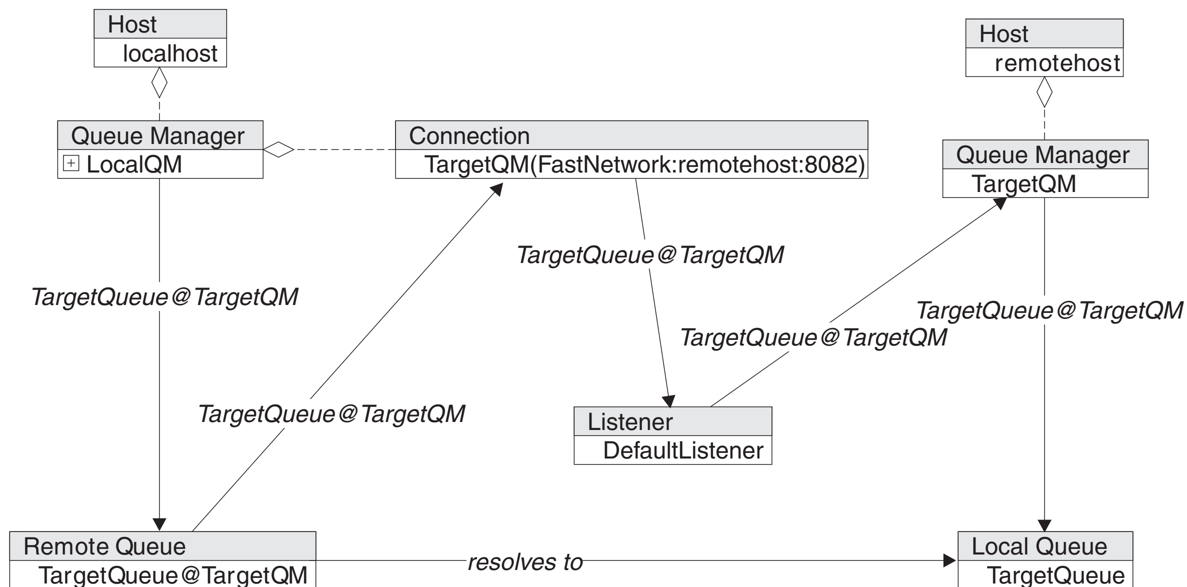


Figure 19. Message resolution for a put

The message route is as follows:

- The message is put on LocalQM addressed to TargetQueue@TargetQM.
- LocalQM performs queue resolution and finds the remote queue reference as an exact match. LocalQM places the message onto the remote queue reference.
- The remote queue reference then performs connection resolution. It looks for a connection that will allow it to pass the message to the queue manager owning the final queue. The remote queue reference finds the connection definition called TargetQM and passes the message to it.
- The connection definition now moves the message to its partner listener, which puts the message to the remote queue manager.
- The remote queue manager performs queue resolution just as if the message had been put locally, finds TargetQueue@TargetQM, and puts the message on it.

Although the connection definition and listener are vital to the message resolution, they do not affect the routing in this example. This is shown in the following diagram:

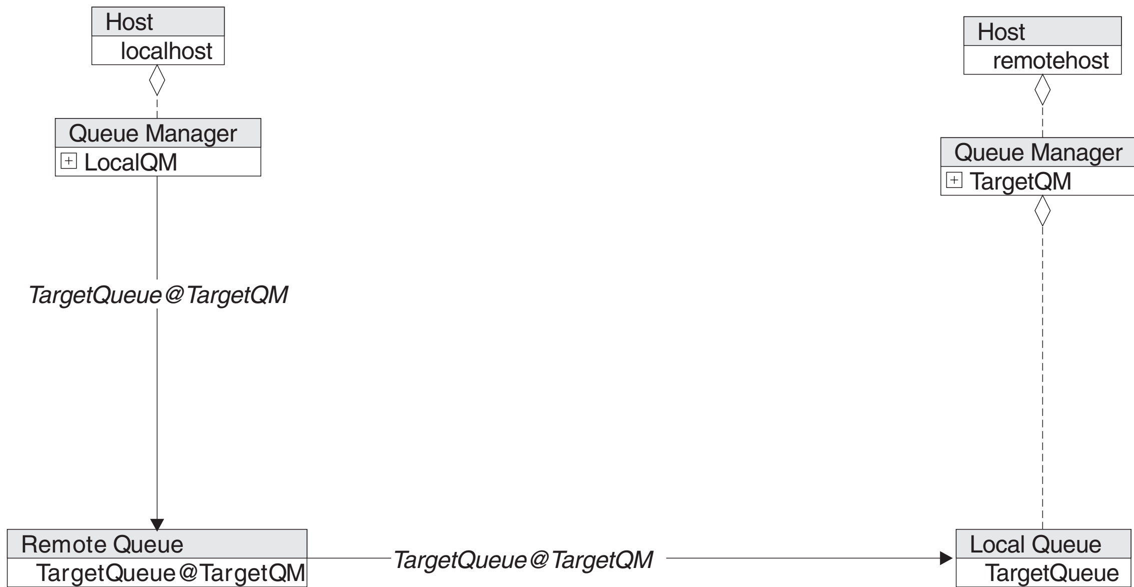


Figure 20. Message resolution for a put

In later examples the connection definitions play a more important role, and they are shown explicitly. For now assume the presence of the logical link formed by the listener and not show them in the diagrams. It is often much more convenient to use a simplified view of the message route. You can do this by thinking of the four elements that contribute to this message resolution as a single, composite, entity. This entity is a Message Route, as shown in the following diagram:

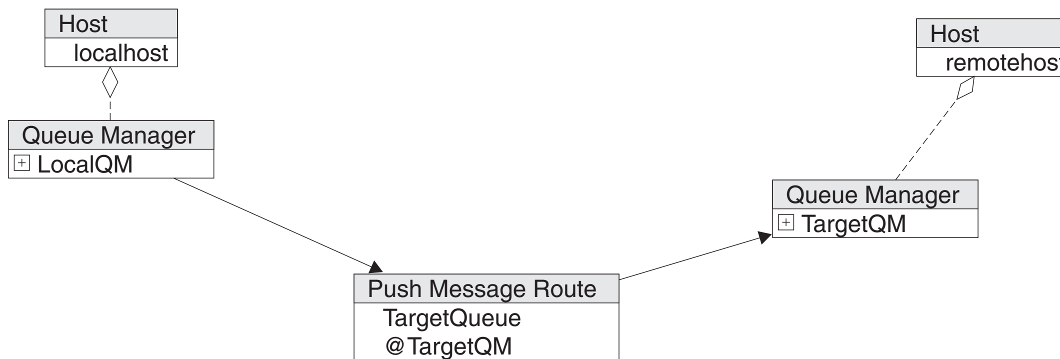


Figure 21. A message route entity

Here you can see the message route that indicates that all messages put to LocalQM and addressed to TargetQueue@TargetQM will be moved directly to the destination. A Message Route is valid only if all the necessary components (Connection Definition, Listener, Remote Queue Definition, and destination queue) are present and correctly configured.

The Message Route is defined as a Push Message Route because messages are pushed from the source queue to the destination queue, by LocalQM.

Aliases on remote queues

You can use aliases on the remote queue, as the last step is simply queue resolution performed on TargetQM. The Queue Alias on the target queue appears to the local system as if it were a queue. The remote queue definition on the local

system is therefore named for the Queue Alias, rather than the target queue. The following diagram makes this clear (note that the connection definition and the listener are hidden):

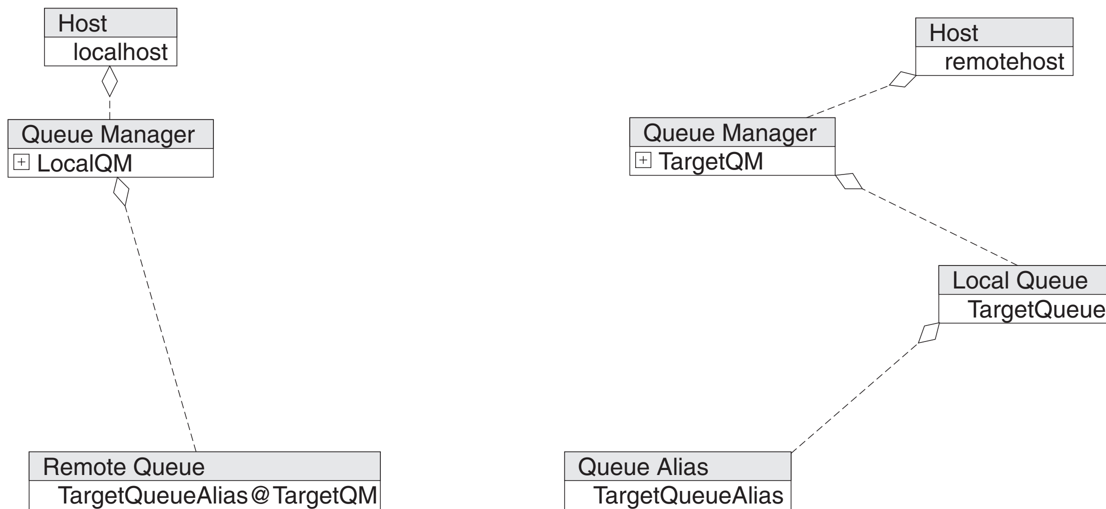


Figure 22. Using aliases on the remote queue

Here a remote queue reference is defined which actually refers to an alias for a queue on TargetQM. When you perform a put on LocalQM addressed to QueueAlias@TargetQM the resolution works as shown in the following diagram:

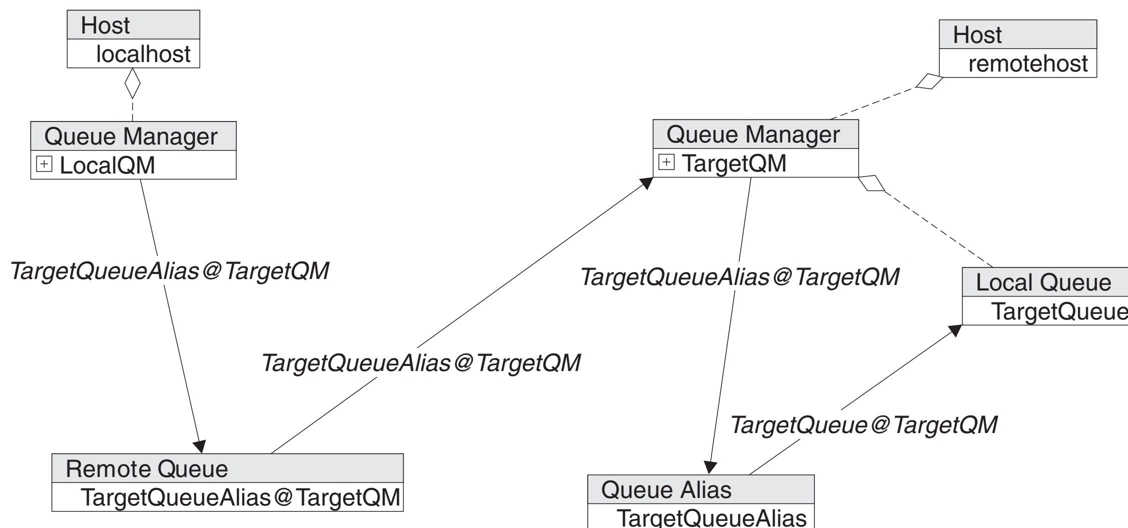


Figure 23. Message resolution for a put to a remote queue, using a Queue alias defined on TargetQM

- Queue resolution on LocalQM finds the remote queue reference. The fact that this is a reference to a queue alias is completely immaterial to queue resolution.
- Connection resolution works entirely as described above
- queue resolution on TargetQM now behaves exactly as local queue resolution of a queue alias described earlier.

Note that the destination name for the message remains QueueAlias@TargetQM until queue resolution onTargetQM. The Remote Queue Definition completes the requirements for another Message Route, as shown in the following diagram:

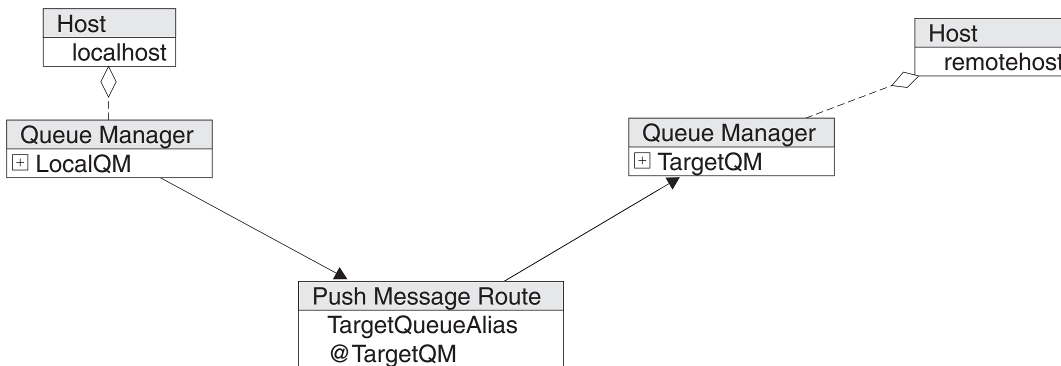


Figure 24. Message route entity of messages put to TargetQueueAlias on TargetQM

Parallel routes

Aliases allow the creation of parallel routes between a source and a destination. This is sometimes useful when you want to send messages synchronously if possible, but asynchronously if the remote end is not currently connected. You can do this with the setup illustrated in the following diagram:

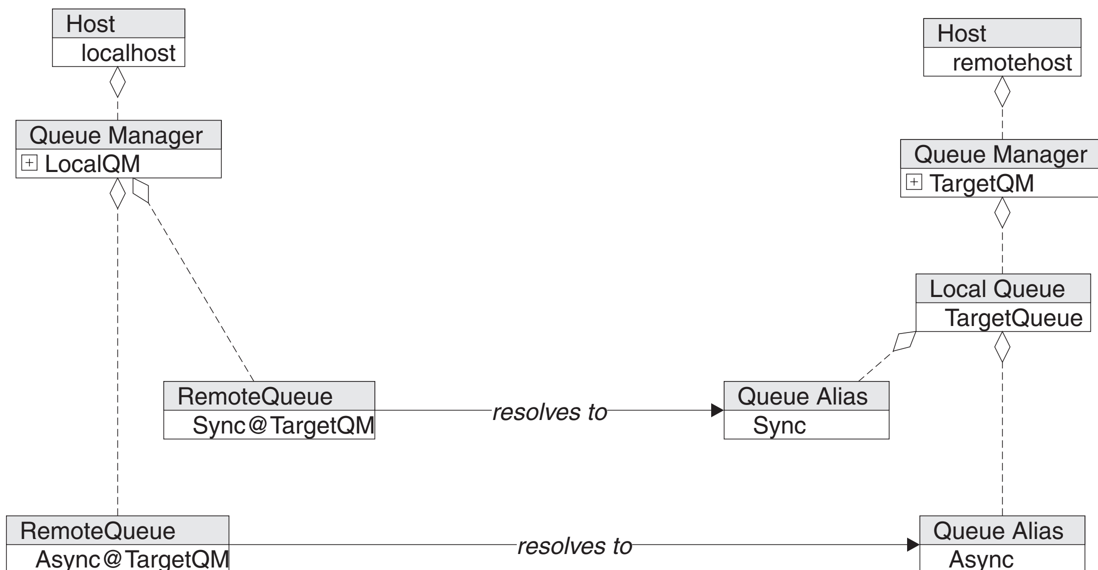


Figure 25. Creating parallel routes between source and destination

Here two aliases have been defined on the target queue. One alias will be used to route synchronous traffic to the target queue, one will be used to route asynchronous traffic.

On LocalQM two remote queue definitions have been defined, one pointing at each alias. You can create an asynchronous Remote Queue Definition called Async@TargetQM, and a synchronous Remote Queue Definition called Sync@TargetQM. By choosing the name of the queue that you put to (Sync@TargetQM or Async@TargetQM) you can choose the route that the message follows, even though the destination is the same. First, the resolution of the

synchronous route by putting a message to Sync@TargetQM, as shown in the following diagram:

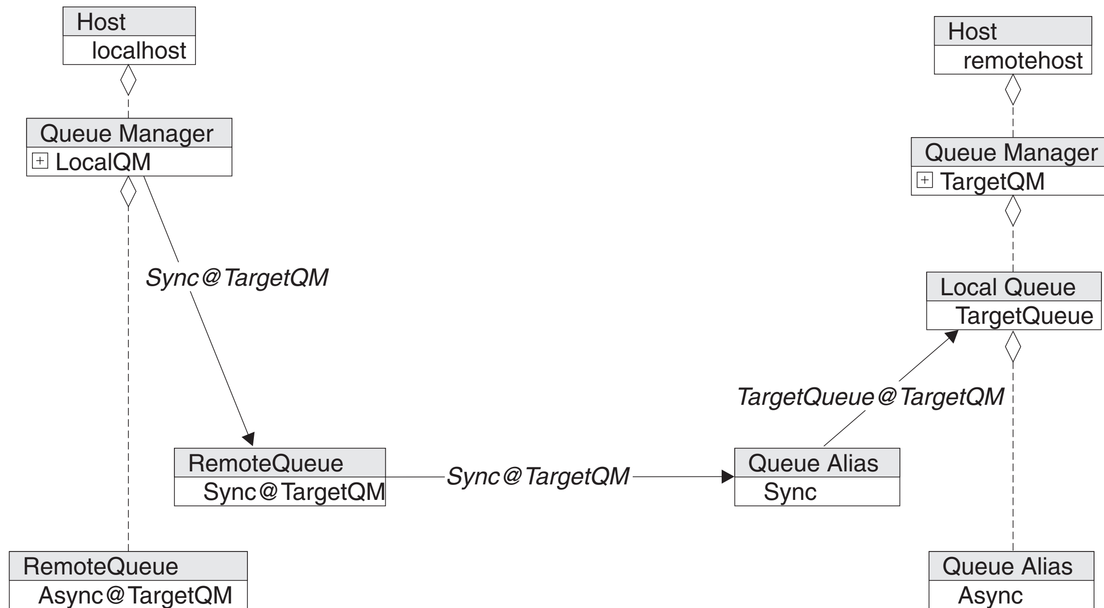


Figure 26. Resolving the synchronous route

And secondly the asynchronous resolution using AsyncAlias@TargetQM, as shown in the following diagram:

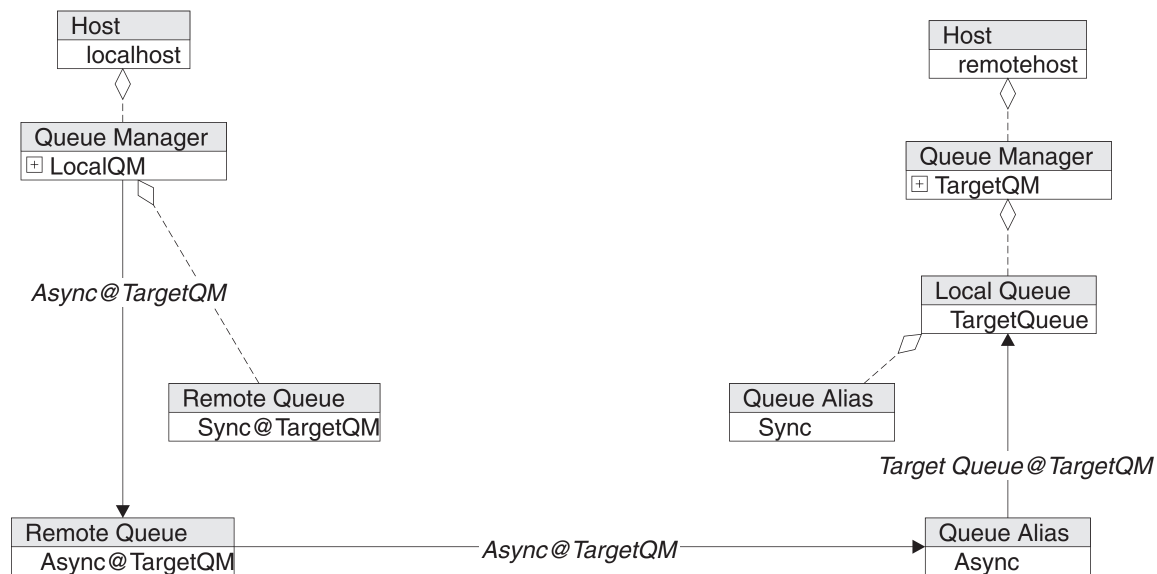


Figure 27. Resolving the asynchronous route

You could choose to view this as a pair of Push Message Routes, as shown in the following diagram:.

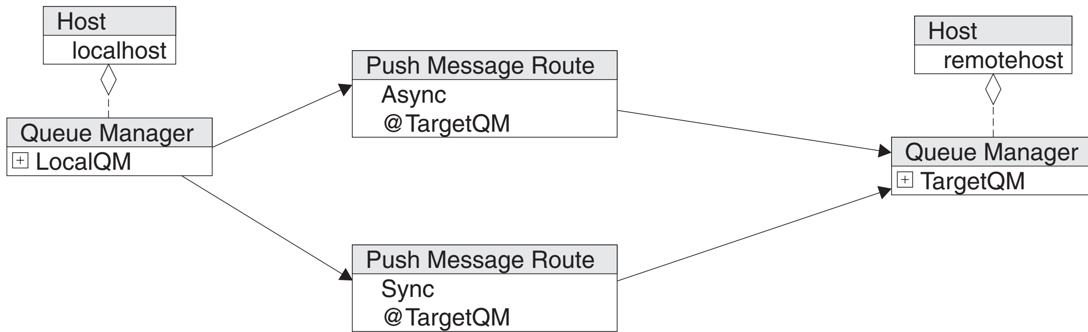


Figure 28. A pair of push message routes

Chaining remote queue references

Remote queue references can be chained together to form a longer route. This requires the use of “Via connections” on page 110.

Pushing store and forward queues

MQe has a queue type that accepts messages on a queue manager basis rather than on a queue basis. These are called Store and Forward (S&F) queues. S&F queues maintain a list of queue manager names, called Queue Manager Entries (QMEs). The S&F queue will accept messages for any queue manager represented by a QME. This acceptance is independent of the destination queue name, and so allows one queue (the S&F queue) to route all messages for a given, or several given queue managers.

S&F queues can operate in two modes, pushing mode and pulling mode. In pushing mode the messages are moved to the next queue manager just as with remote queue references. In pulling mode the messages are removed from the S&F queue by the action of a Home Server Queue. This section deals only with the pushing of messages, pulling messages with a home server queue is described in another section. A typical pushing S&F queue system might look like this:

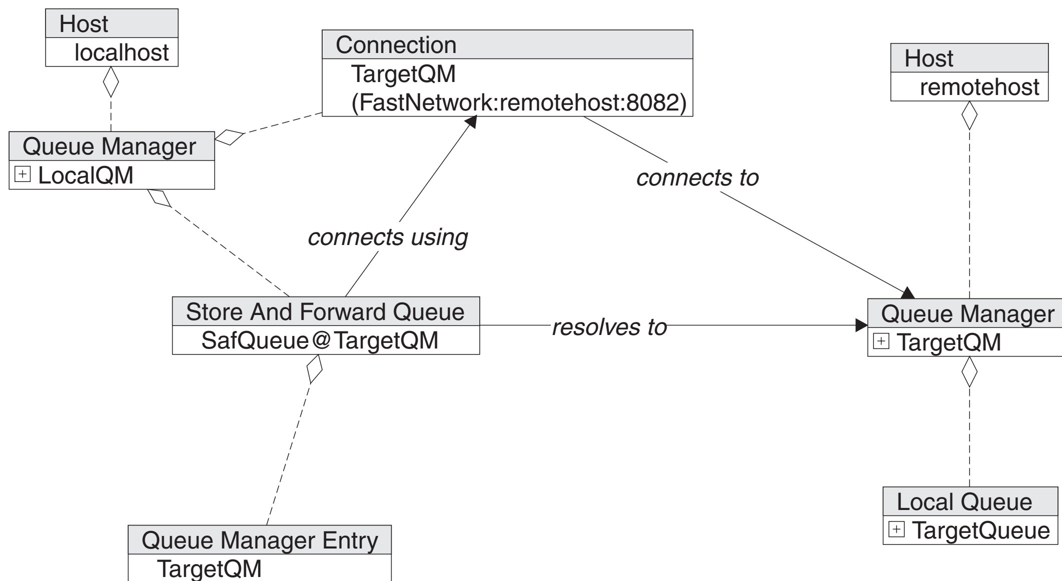


Figure 29. A typical pushing S&F queue system

A S&F queue called SafQueue has a queue manager entry (QME) for TargetQM. This allows it to accept messages for any queue on TargetQM. In common with ordinary Remote Queues, a Store and Forward queue requires a connection definition/listener pair set up in order to push messages. Unlike a normal Remote Queue Definition, a Store and Forward Queue effectively pushes to a Queue Manager rather than to a queue. The message arrives at the Queue Manager, where queue resolution is performed. When a message is put to LocalQM addressed to TargetQ@TargetQM the resolution is as follows:

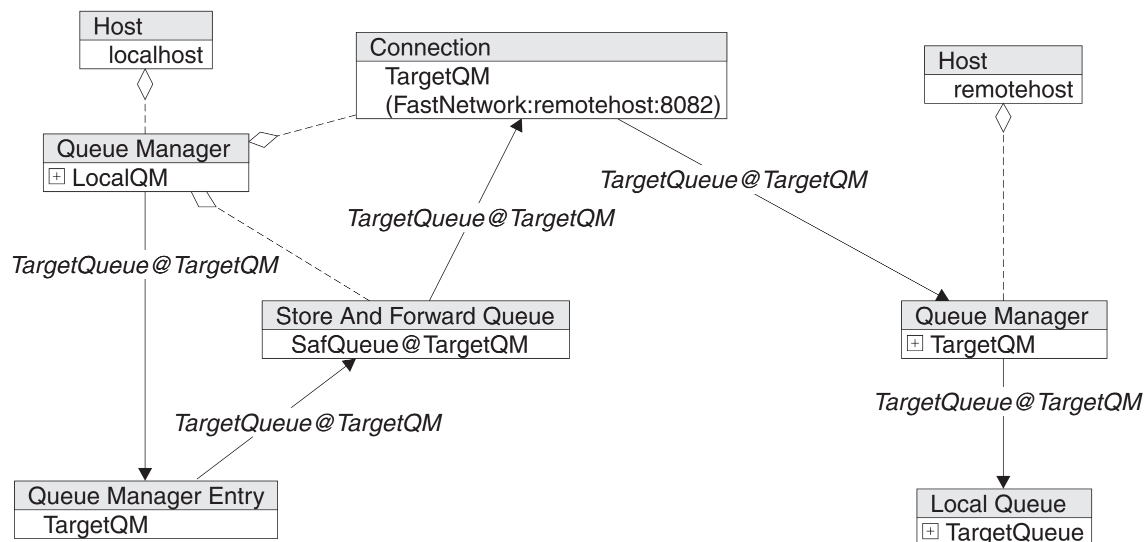


Figure 30. Routing of a message put to LocalQM and addressed to TargetQ@TargetQM

- LocalQM performs queue resolution which finds the queue manager entry TargetQM on SafQueue. LocalQM puts the message to the QME.
- Putting a message to the QME is equivalent to putting the message on the S&F queue owning the QME.

- The S&F queue performs connection resolution and finds the connection definition, and so uses it to push messages to RemoteQM.
- The queue manager then performs queue resolution and places the message on the target queue.

The Store and Forward queue forms part of a Multi Message Route. This abstract entity represents the potential for messages addressed to any queue on TargetQM, and so is called *@TargetQM, as shown in the following diagram:

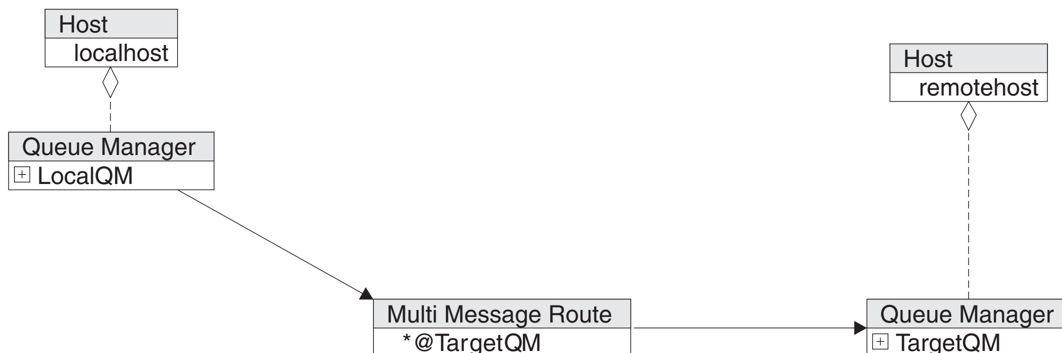


Figure 31. A multi message route

If there is no queue to which the message can be put, then it is not delivered. This prevents any further messages from being pushed from that Store and Forward queue to that Queue Manager.

S&F queues and remote queue references

Because Store and Forward (S&F) queues can accept messages for any queue on a given queue manager, they can appear to be in conflict with a remote queue reference. In such cases the remote queue reference takes precedence, because it is more specific. So if add a remote queue reference to the S&F queue resolution, the message route resolution changes immediately, and the S&F queue becomes irrelevant, as shown in the following diagram:

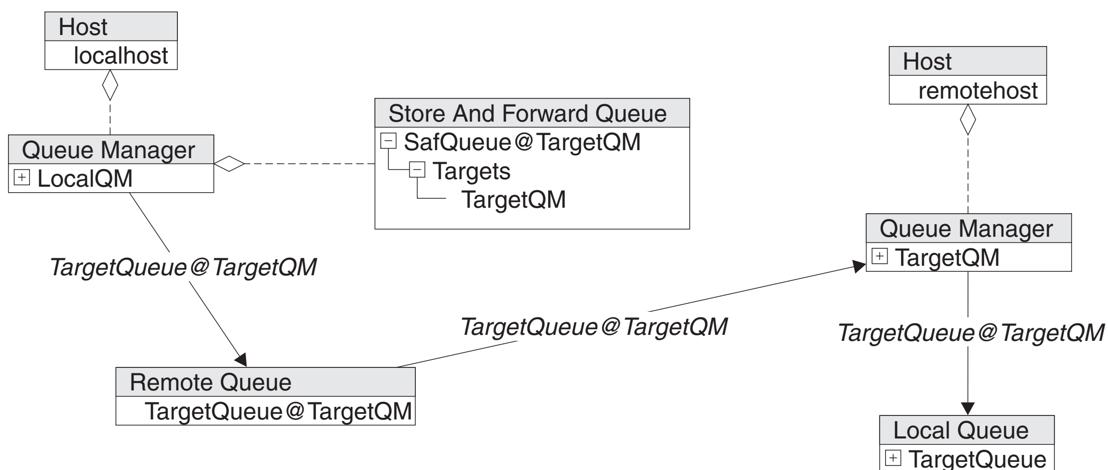


Figure 32. How routes using remote queue definitions take precedence over store-and-forward queue routes

The queue resolution finds the best (most exact) match for the message address.

So a message put to QueueAlias@TargetQM goes via the S&F queue (asynchronous transmission), but a put to TargetQueue@TargetQM goes synchronously via the remote queue reference.

Chaining S&F queues

Pushing store and forward queues can be chained together into a more complex route, as shown in the following diagram:

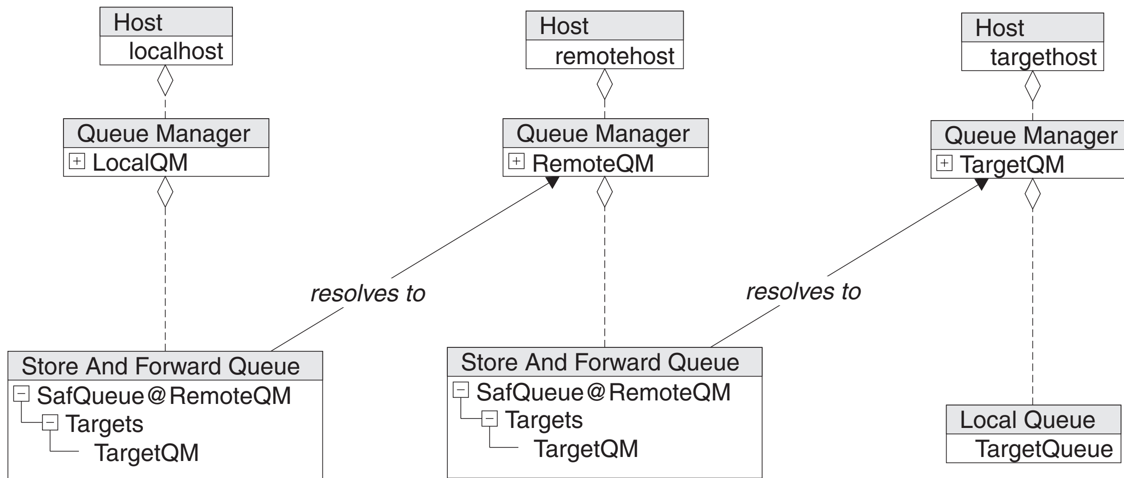


Figure 33. Pushing S&F queues chained together

The Store and Forward queue on LocalQM (SaFQueue@RemoteQM) has a Queue Manager Entry for TargetQM, but actually pushes to RemoteQM. LocalQM requires a connection definition to RemoteQM, but not to TargetQM. A message can then be transported via the intermediate S&F queue, as shown in the following diagram:

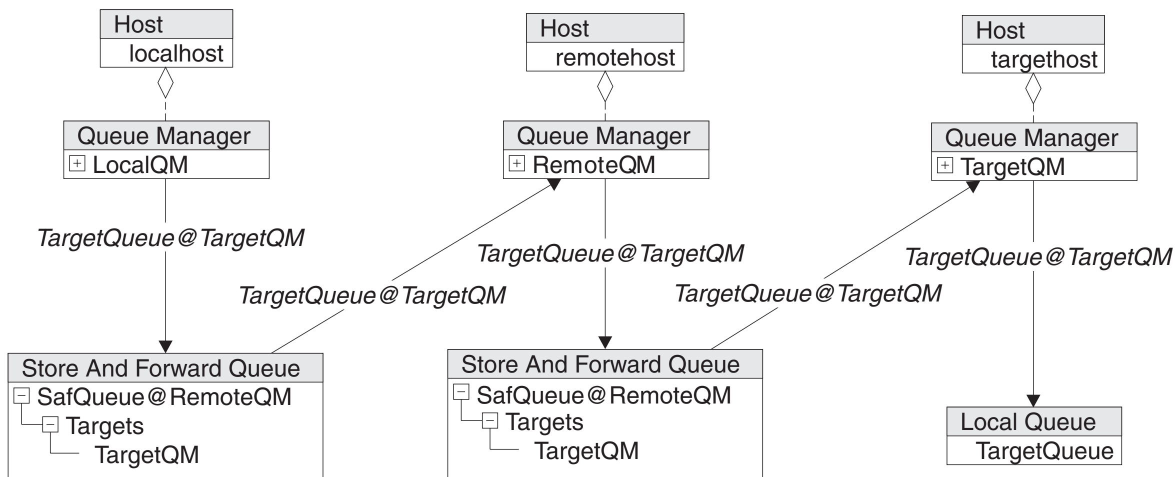


Figure 34. Transporting messages via an intermediate S&F queue

This works because the combination of queue resolution and connection resolution on LocalQM results in the message being put to the S&F queue on RemoteQM, which can then move it to its destination. The chain of Store and Forward Queues could be arbitrarily long, with each queue manager in the chain needing to know

only about the next queue manager in the chain. The Message Routes express this very succinctly, as shown in the following diagram:

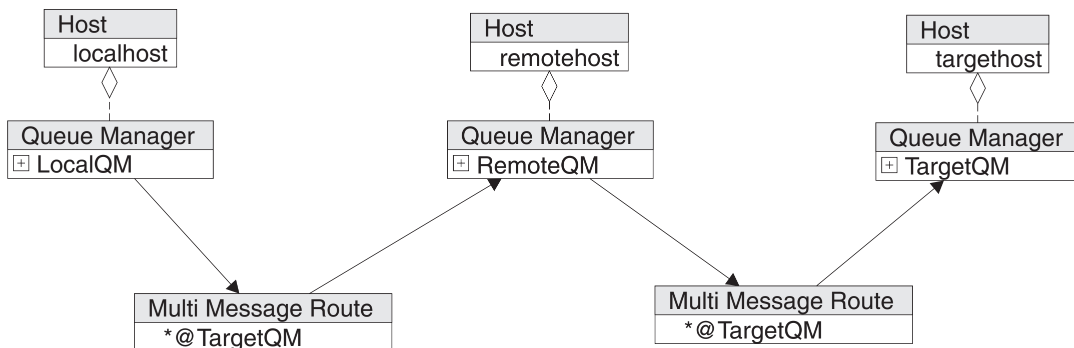


Figure 35. A chain of store and forward queues

Home server queues

Home server queues pull messages from store and forward queues. The S&F queue may be a 'pushing' S&F queue (that is, has a valid connection definition). Home server queues only pull messages across a single 'hop', and only pull messages whose intended destination is the local queue manager - the queue manager upon which the home server queue resides. A typical Home Server Queue configuration is illustrated below:

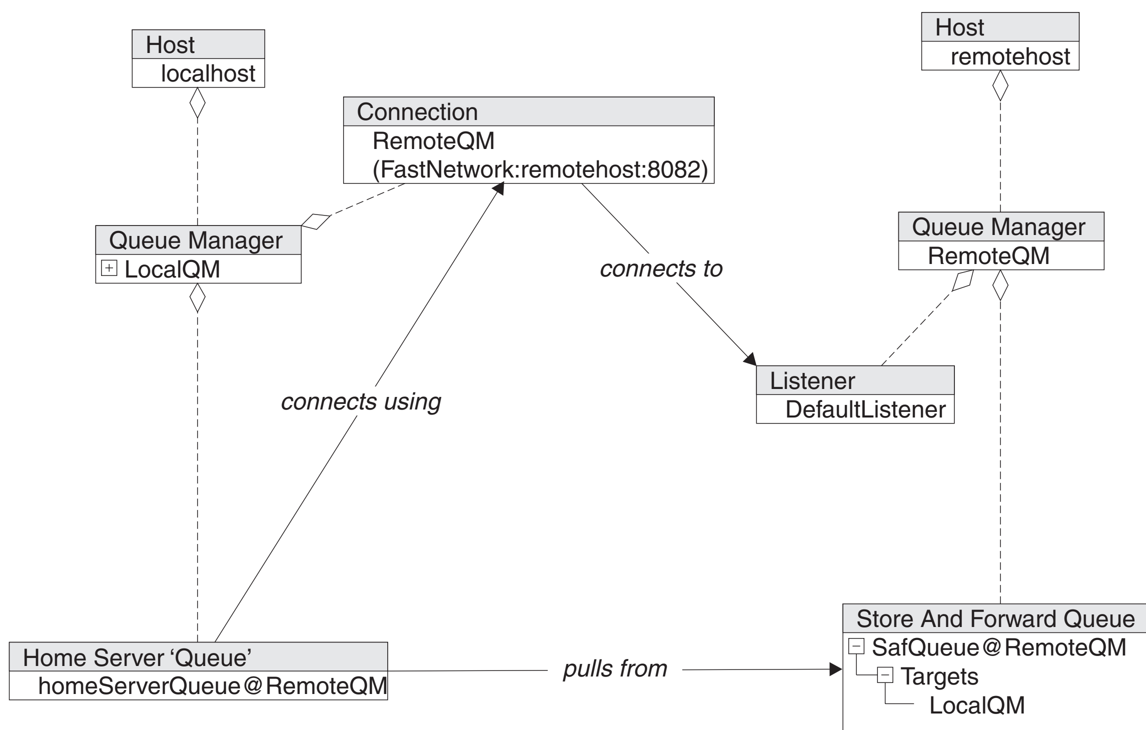


Figure 36. A home server queue configuration

The diagram shows a simple HomeServerQueue setup. In this configuration the server queue manager has no connection definition to the client; instead it has a

store queue (that is, a store and forward queue with no target queue manager) that collects all messages bound for the client. This message collection embraces all queue destinations on the client.

The client pulls the messages from the store queue using a home server queue pointing at the store queue on the client. The home server queue never stores messages itself, it collects them from the store queue and delivers them to their destinations on the client. The client makes the connection request to the server using its connection definition.

The home server queue 'homeServerQueue@RemoteQM' attempts to pull messages from the queue manager 'RemoteQM'. It requires a connection definition to be able to do this. The home server queue is able to pull messages only if there is a store and forward queue that is storing messages for LocalQM.

Messages that are pulled from RemoteQM are then 'pushed' to local queues on LocalQM. This is shown in the following diagram, where a Home Server Queue on LocalQM is pulling messages (for LocalQM) from RemoteQM. In this case a message for TargetQueue@LocalQM is shown being pulled, and the resolution at the queue manager has been hidden for clarity. In reality, the Home Server Queue presents each pulled message to the local queue manager for resolution, as shown in the following diagram:

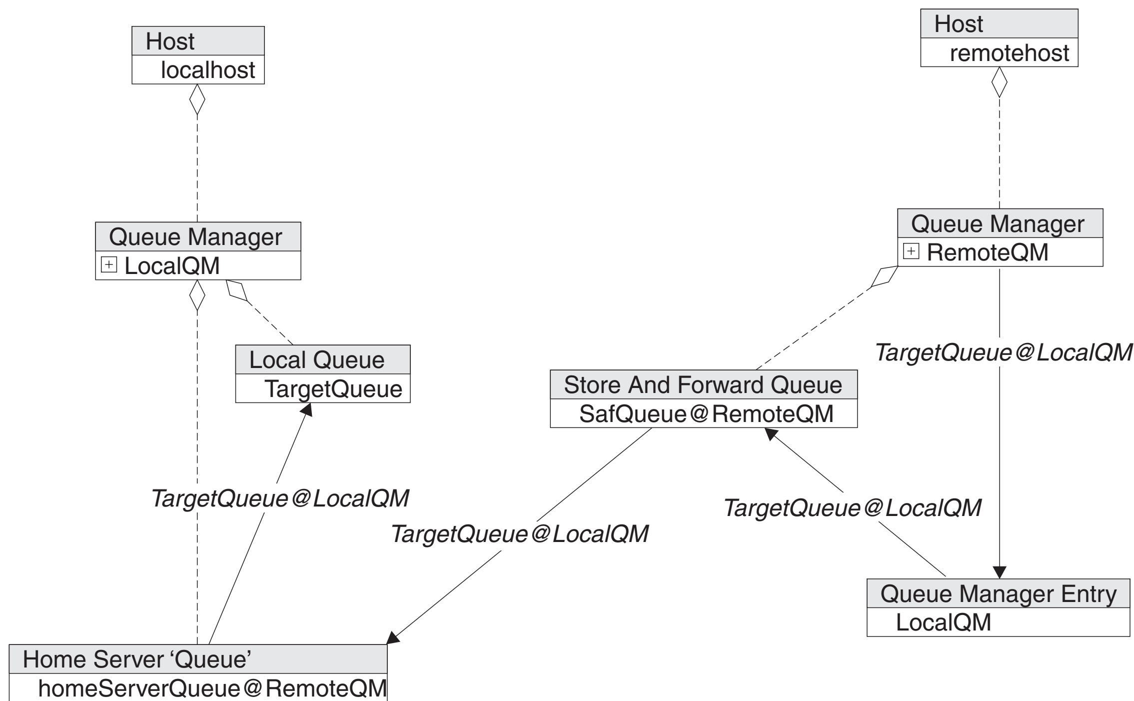


Figure 37. A home server queue pulling messages

The pull message route can be viewed at a more abstract level, as shown in the following diagram:

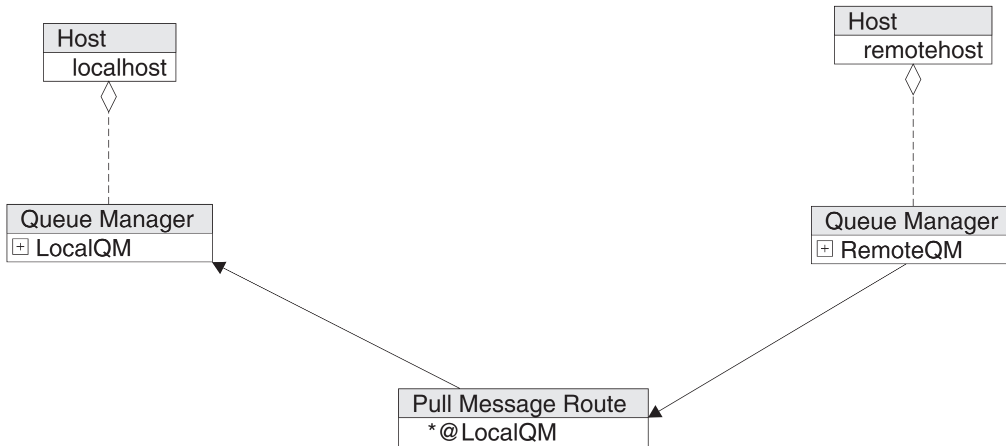


Figure 38. An abstract pull message route

How are pulled message routes useful, and where would you use them? The most important feature of a pulled message route is that the flow of messages is under the control of the local queue manager. This makes it very useful to a client that spends much of its time disconnected. If you had to rely on the server pushing message, the server would need to continuously poll the client to check if it was available. This would not be a good solution for large numbers of clients, as much of the servers time would be spent polling for disconnected clients.

Instead, with a Home Server queue, each client pulls messages when it is connected, and the server only has to deal with real requests from connected clients. One concrete example of this is the administration of queue managers that do not have listener capability. Administration messages for the client are placed upon a Store and Forward queue. The client can then use a Home Server queue to pull these when it is connected. Administration reply messages could then be pushed using normal push remote queue, as shown in the following diagram:

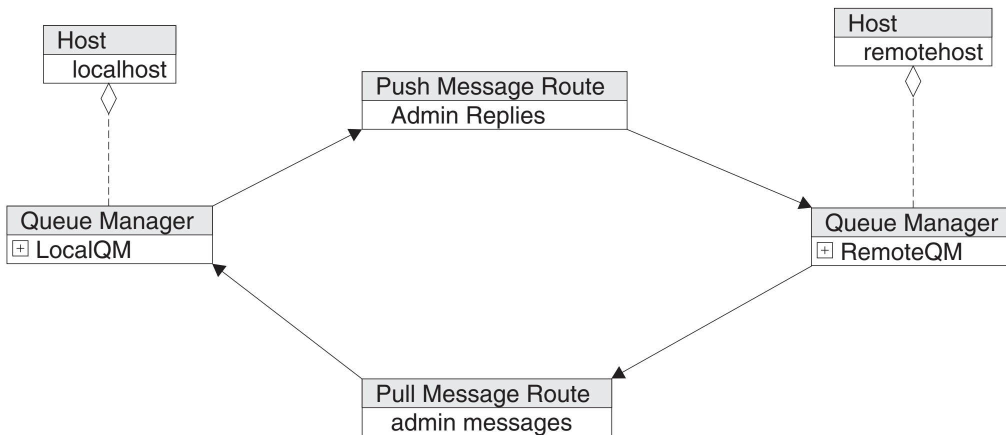


Figure 39. Administering queue managers that do not have listener capability

Via connections

Via connections allow messages to be routed via an intermediate queue manager. For example, you might want messages from LocalQM to travel to TargetQM via

RemoteQM. You can already do this with 'pushing' store and forward queues, but via connections provide another mechanism, as shown in the following diagram:

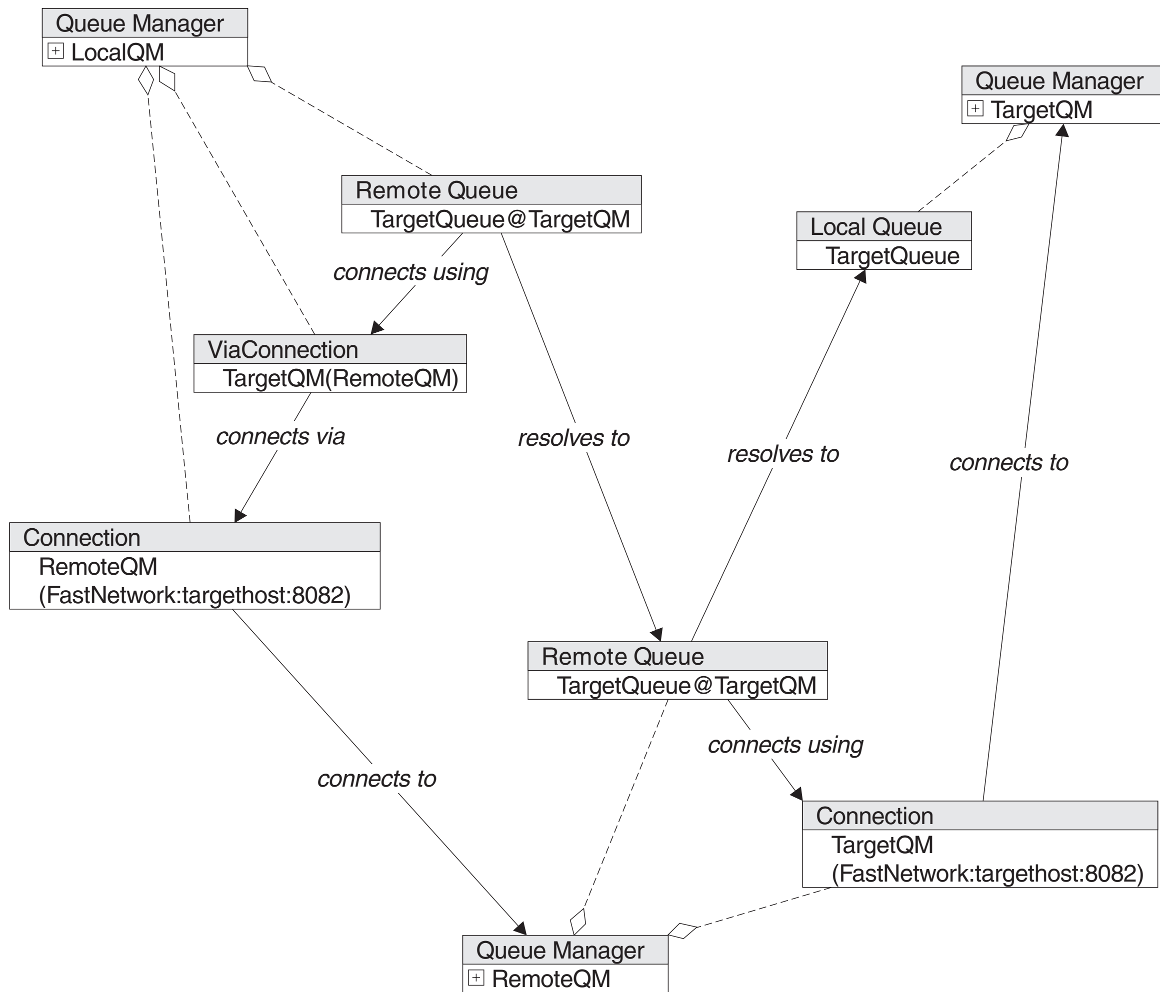


Figure 40. Via connections

The diagram above illustrates the components being used. The connection definition called 'TargetQM' on LocalQM does not contain the address of TargetQM, but simply refers to the connection definition called 'RemoteQM'. This means that any messages destined for TargetQM will be sent to RemoteQM, and RemoteQM will be able to move the messages onward. In the diagram above, RemoteQM has the necessary connection to move the message to TargetQM.

The message flows as expected, as shown in the following diagram:

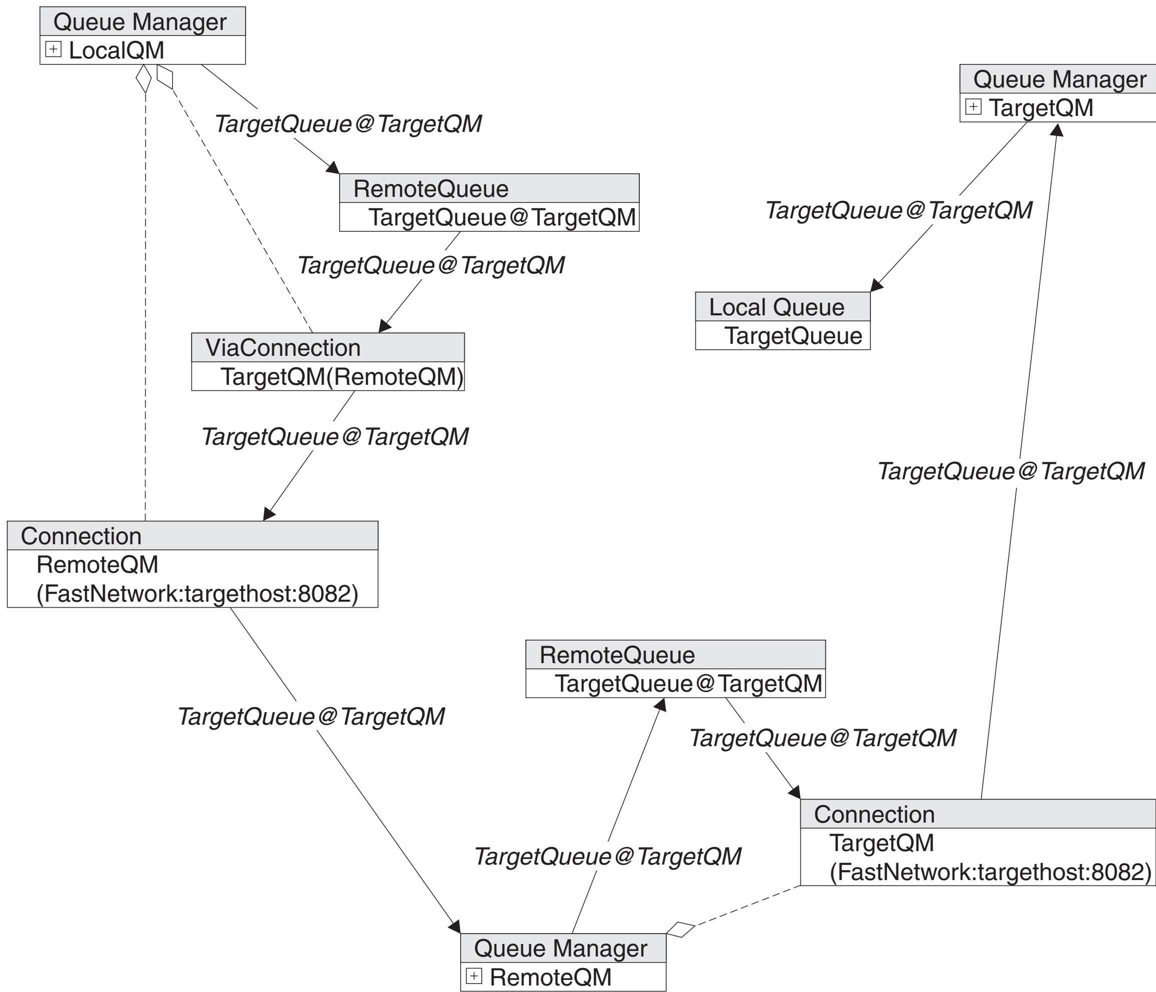


Figure 41. Message flow using a via connection

The Remote Queue on LocalQM uses Connection Resolution to find the Via Connection. This then passes the message on to the real connection which moves the message to RemoteQM. On RemoteQM queue resolution proceeds as for the simple case.

You can see the topology most clearly using Message Routes, as shown in the following diagram:

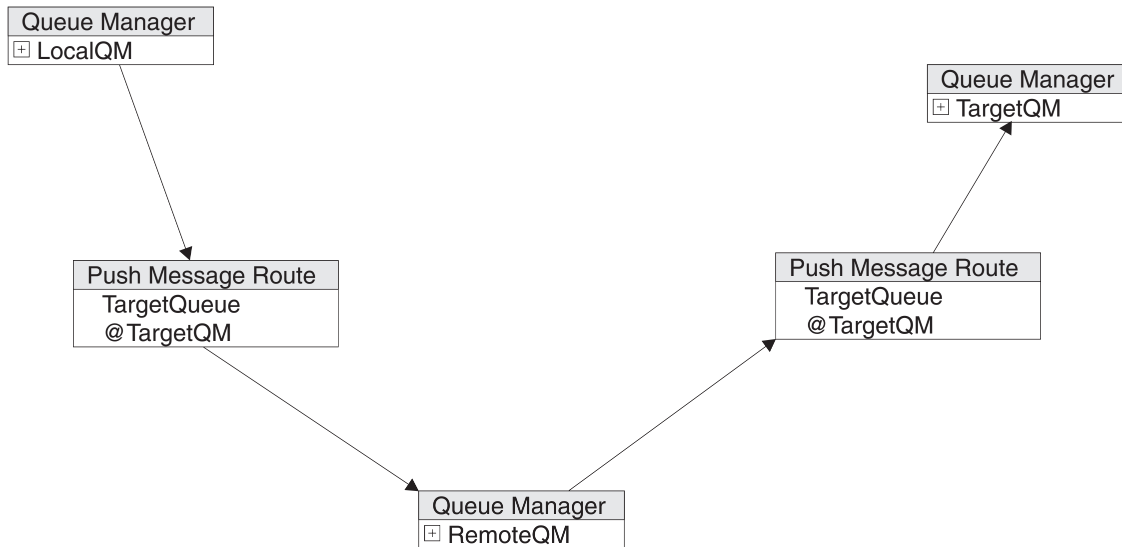


Figure 42. Via connections expressed using message route schema

This is known as 'chaining remote queues'. The central remote queue can be synchronous, asynchronous, or even a store and forward queue.

Rerouting with queue manager aliases

Fail-over is a common situation that illustrates the important part that Queue Manager Aliases play in routing.

In the following examples, you can see a client communicating with a server, and a have a backup server that can be used if the main server fails, or is taken down for maintenance:

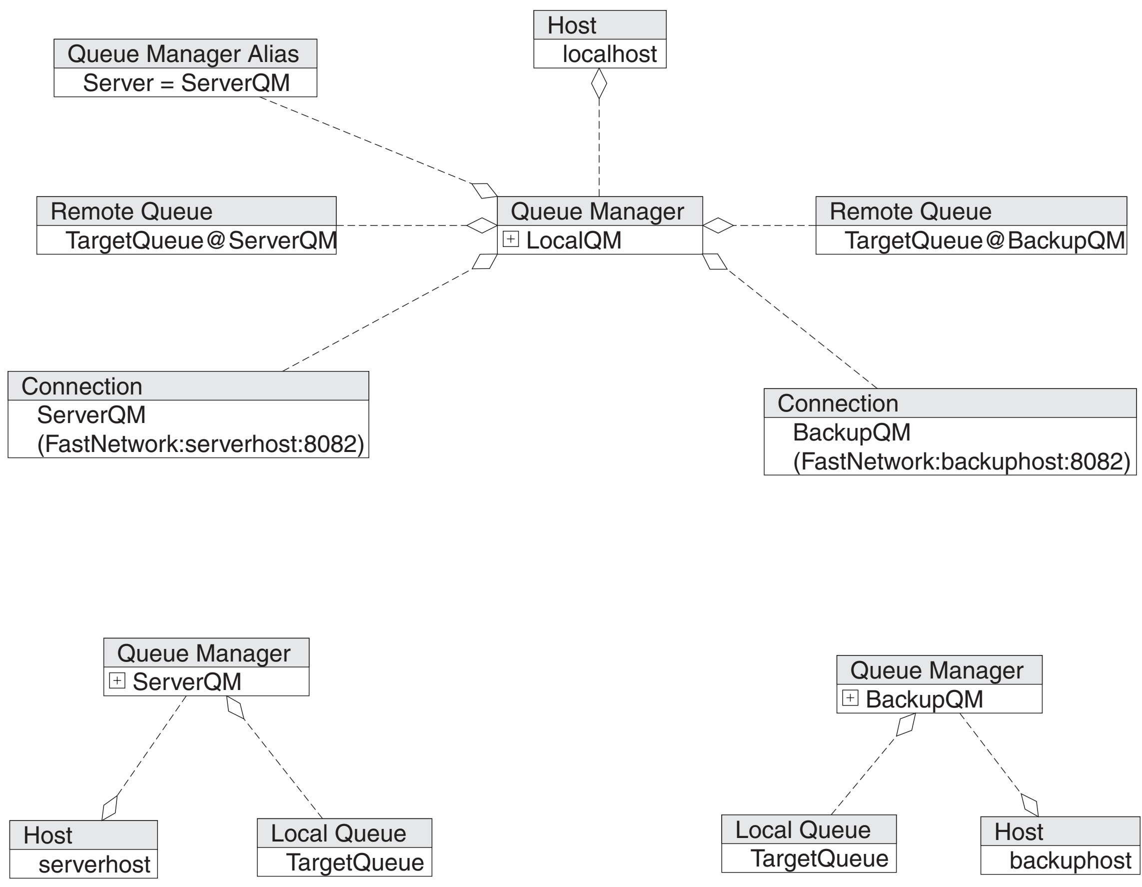


Figure 43. Queue manager aliases and fail-over.

The diagram above shows the local client queue manager, with a connection to ServerQM and a remote queue definition for TargetQueue@ServerQM. The server (bottom left) has a local queue as the target for the example message, and this is mimicked by the backup server (bottom right). Additionally, on the client queue manager, there is a Queue Manager Alias mapping the name Server to ServerQM. This mapping is then used for messages put to the server. The message resolution is shown below for the normal operating configuration, where a message put to TargetQueue@Server is directed to TargetQueue@ServerQM:

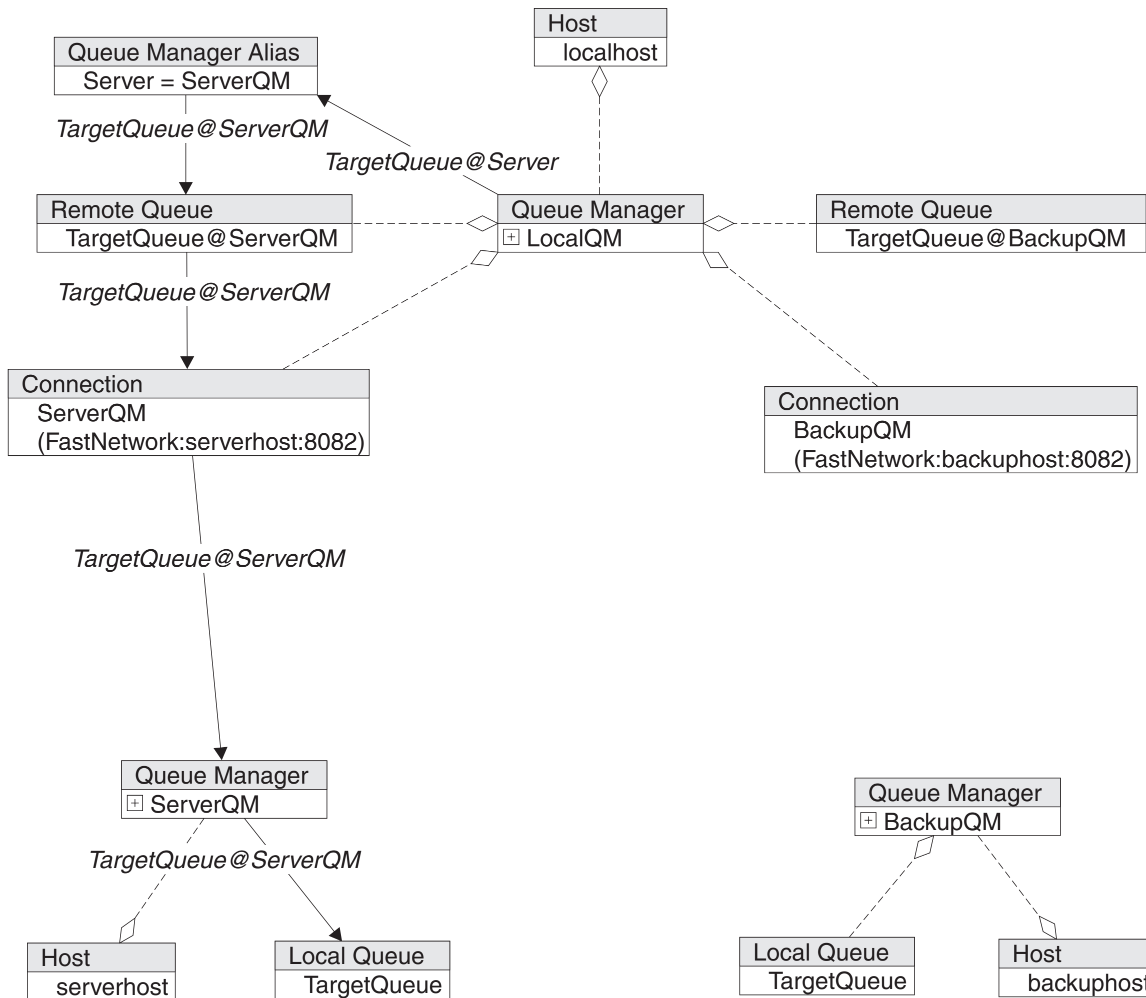


Figure 44. Routing traffic using a "server" alias

The alias maps messages for Server to ServerQM, and this selects the remote queue definition TargetQueue@ServerQM. If the network administrator needs to route traffic to the backup server, only the Queue Manager Alias needs to be changed (it is in fact deleted, and recreated with a different target name, in this case BackupQM):

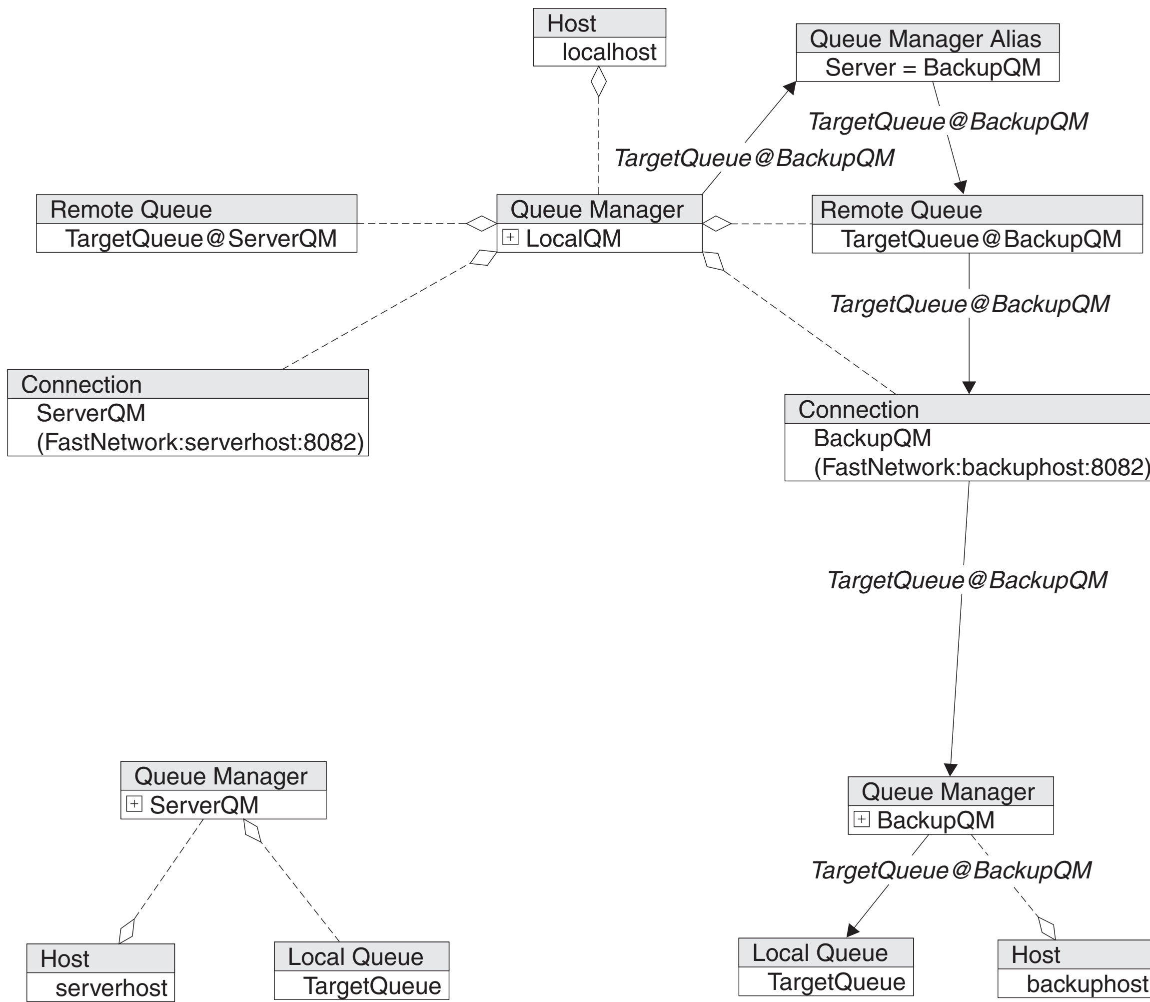


Figure 45. Routing traffic to the backup server, using a "server" alias

The change of alias reroutes the message to a different remote queue, and hence on to the backup queue manager and to TargetQueue@BackupQM. There is a pair of message routes, one to each server, and a Queue Manager Alias to choose between the message routes, as shown in the following diagram:

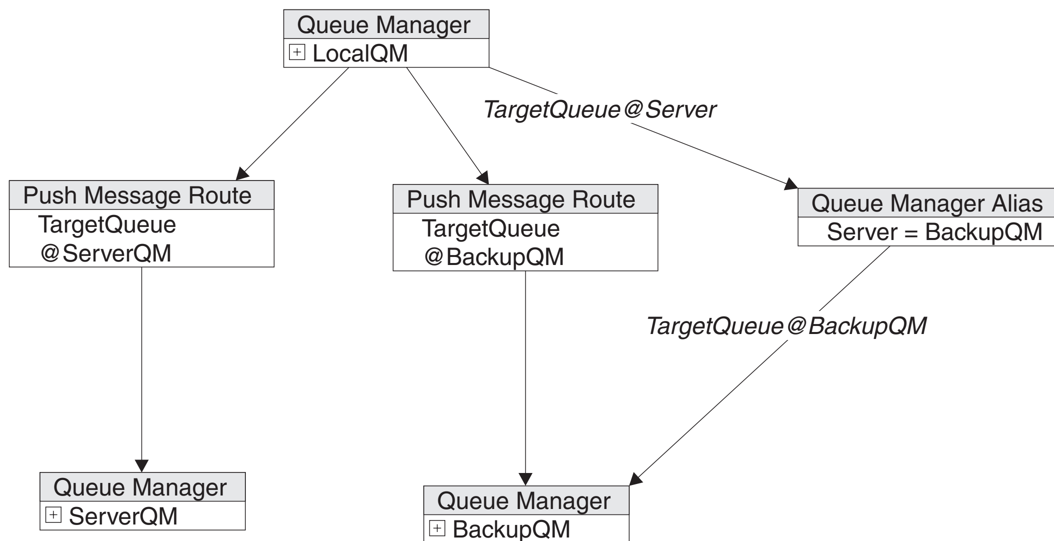


Figure 46. Choosing between message routes

The example above required a change to every client on a system that requires rerouting to a backup server. If there are a large number of clients this might be impractical. In addition, each client requires two complete message route definitions (a remote queue and a connection definition for each). You can avoid the need to change the client by having a second server ready to listen on the same address and port as the first. When the administrator wants to change over the first can be brought down, and the second can change over. In this situation it might be useful to keep the names of the servers different. The backup server can be given a Queue Manager Alias mapping BackupQM to ServerQM. This allows BackupQM to take the place of ServerQM.

MQe-MQ bridge message resolution

A connection between MQe and MQ queue managers involves a collection of objects. The following diagram shows only the entities that form the communications link between the two queue managers:

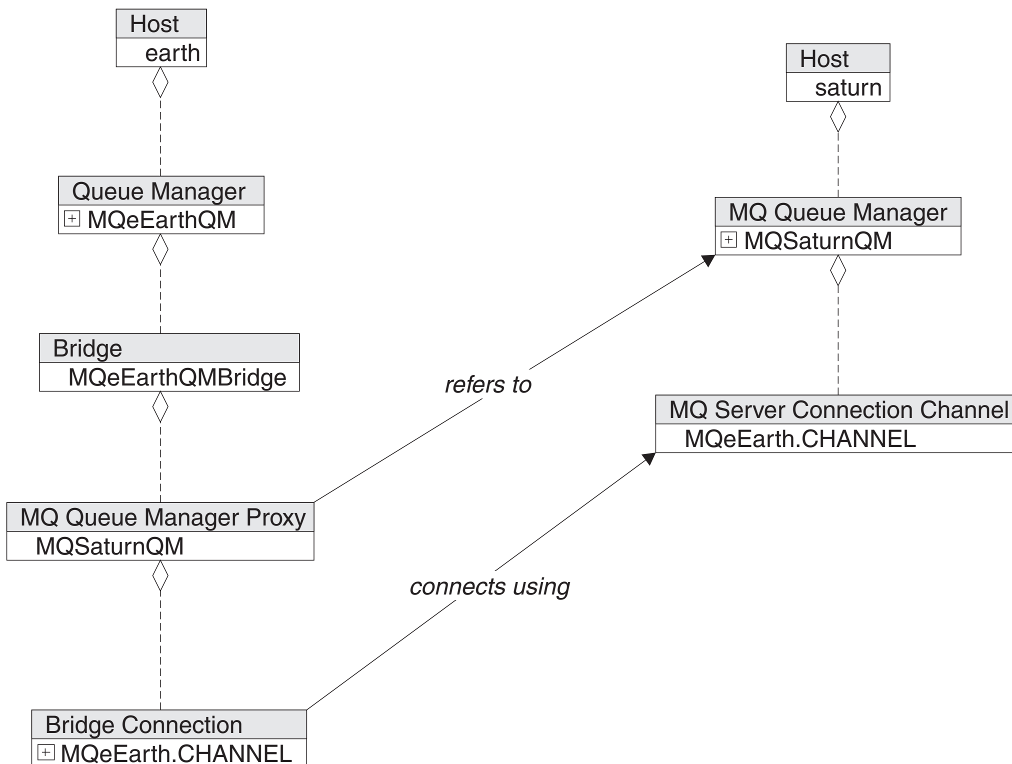


Figure 47. Connecting MQe and MQ queue managers.

The important entities are:

- (Bridge)MQeEarthQMBridge - a bridge resource owned and controlled by the MQeEarthQM queue manager.
- (MQ Queue Manager Proxy)MQSaturnQM - describes MQSaturnQM and how to connect to it.
- (BridgeConnection)MQeEarth.CHANNEL - a communications path between MQeEarthQM and MQSaturnQM.
- (MQ Server Connection Channel) MQeEarth.CHANNEL - a standard MQ server channel providing an entry point to MQSaturnQM for MQeEarthQM.

These entities are described in more details in other parts of this documentation. These entities are used in the following examples of bridge connectivity, but are not shown in the diagrams.

Pulling messages from MQ

By setting up a Transmit queue on MQ, and a bridge listener on an MQe queue manager, you can enable the queue manager to pull messages from the transmit queue. Although in theory this is sufficient to pull messages from the transmission queue, you cannot place messages onto the transmission queue without creating extra queues on an MQ queue manager.

Single pull route:

To allow the messages to be correctly routed, you can create extra queues on an MQ queue manager. The simplest form is to create a remote queue on MQ to allow

messages addressed to TargetQueue@MQeEarthQM to be accepted by the MQ queue manager, as shown in the following diagram:

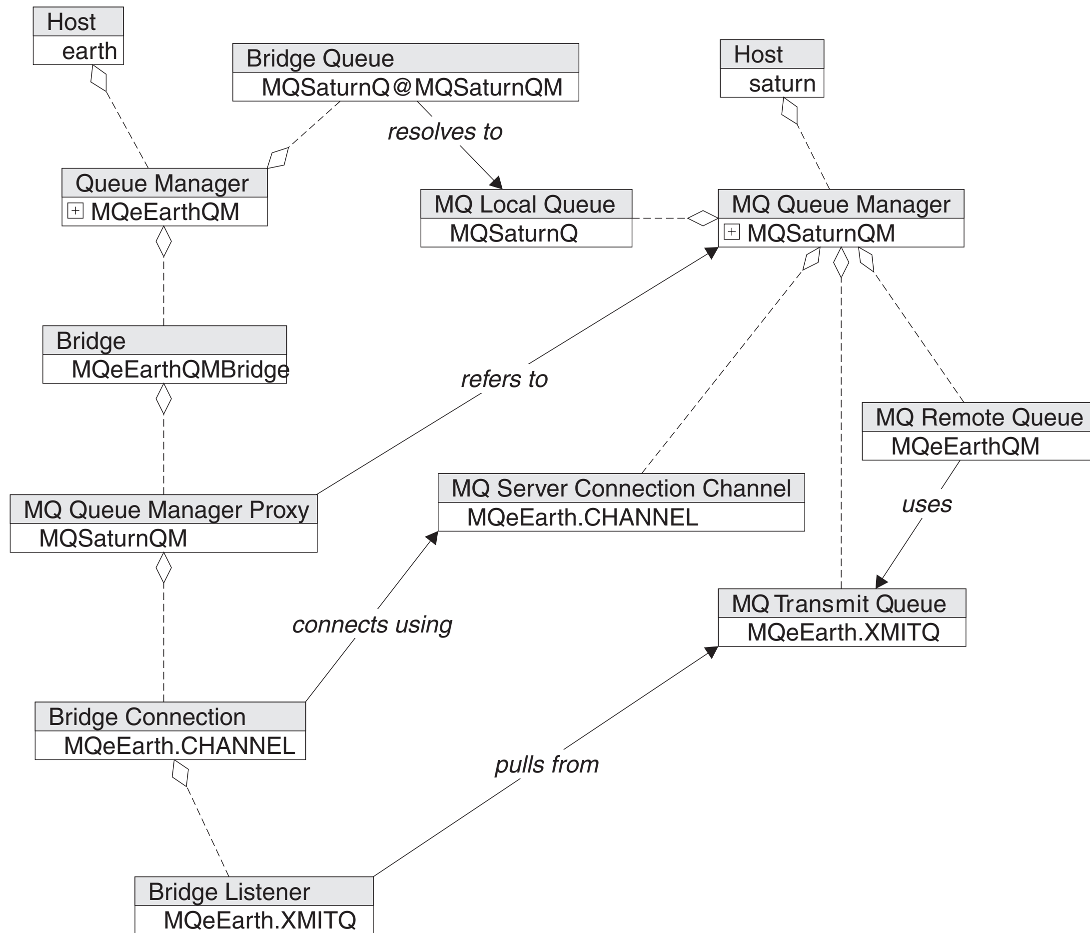


Figure 48. Creating a remote queue on MQ

Messages addressed to TargetQueue@MQeEarthQM are placed upon the MQ Transmit queue. The bridge listener then pulls them from the transmit queue and presents them to the MQe queue manager. Message resolution then takes place, as shown in the following diagram:

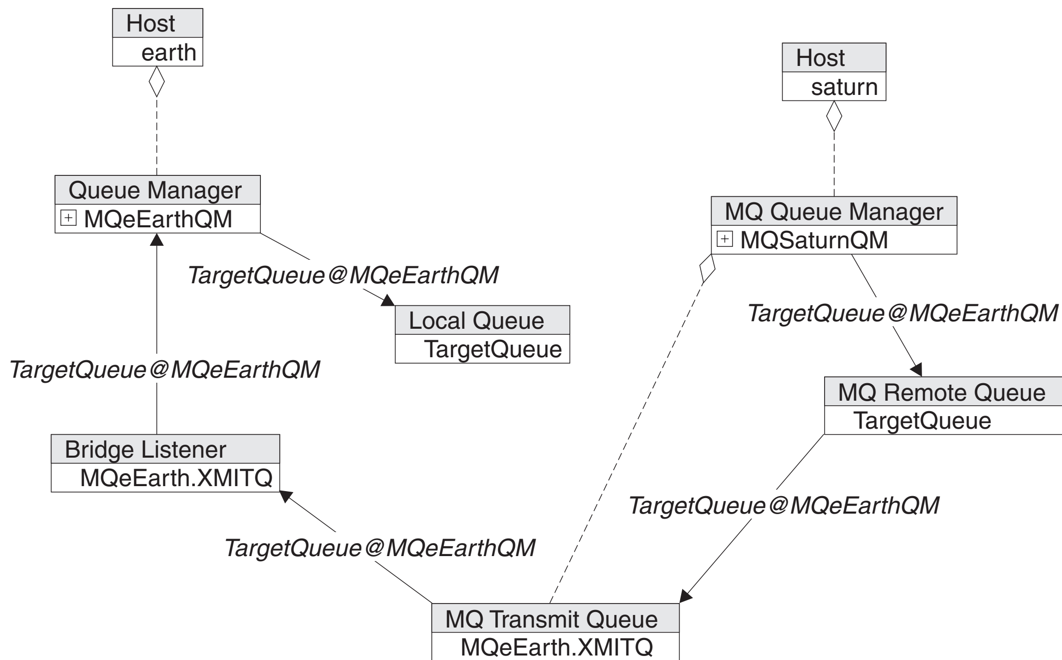


Figure 49. Bridge listener pulling from an MQe transmit queue

This is effectively a single pull message route:

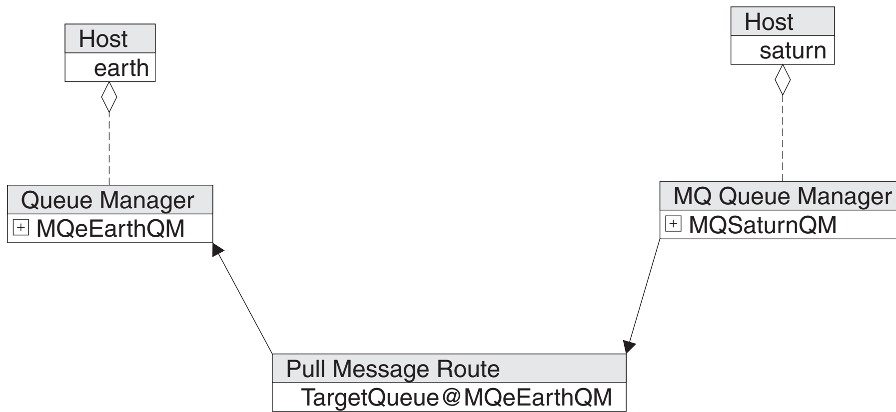


Figure 50. A single pull message route

Multiple pull route:

It is generally more efficient to use a multiple pull message route as this requires the same number of resource definitions, but will handle all the traffic for the MQe queue manager. This is done using a Remote queue manager alias on MQ (effectively a remote queue where the target queue name is the same as the target queue manager name), as shown in the following diagram:

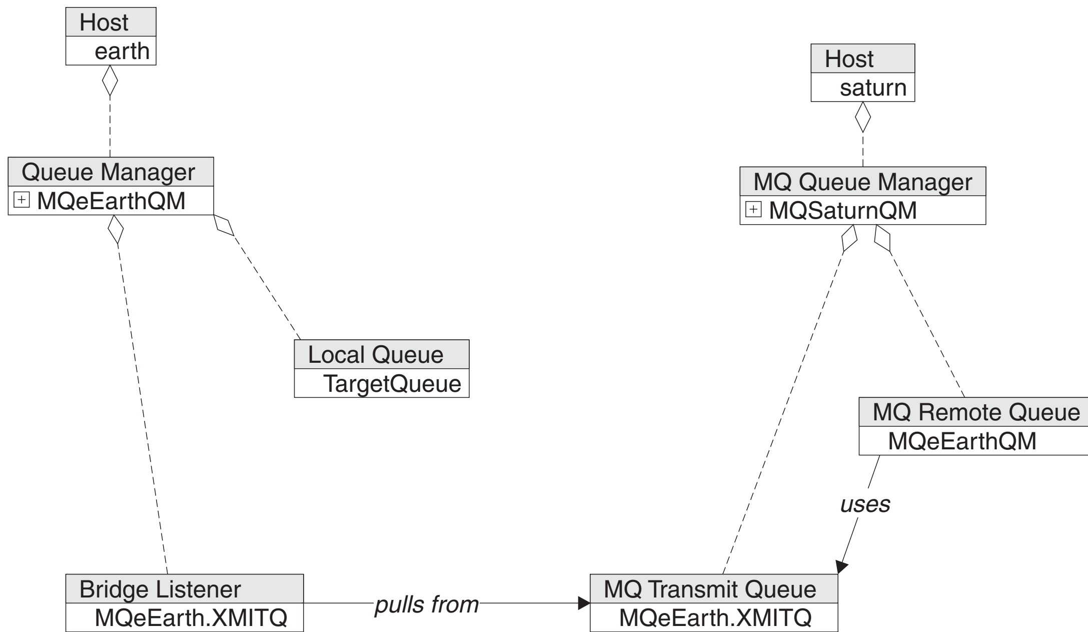


Figure 51. A multiple pull message route

Message resolution works as before, but now messages for any queue on MQeEarthQM will be moved, making this a multiple pull message route, as shown in the following diagram:

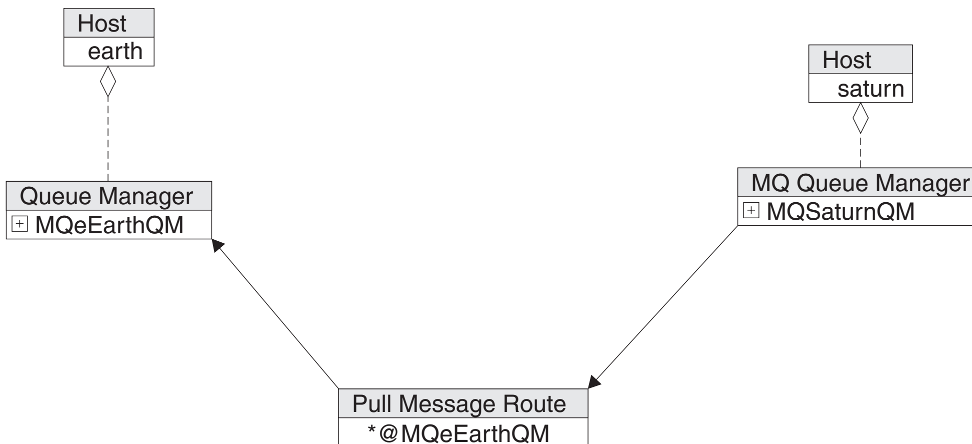


Figure 52. Multiple pull route, expressed using message route schema

Pushing messages to MQ

Pushing messages to MQ is quite straightforward. Again you need to presume the presence of the common components described in “MQe-MQ bridge message resolution” on page 117, but now you need to create a Bridge Queue which is an MQe Remote queue that refers to a queue on an MQ queue manager, as shown in the following diagram:

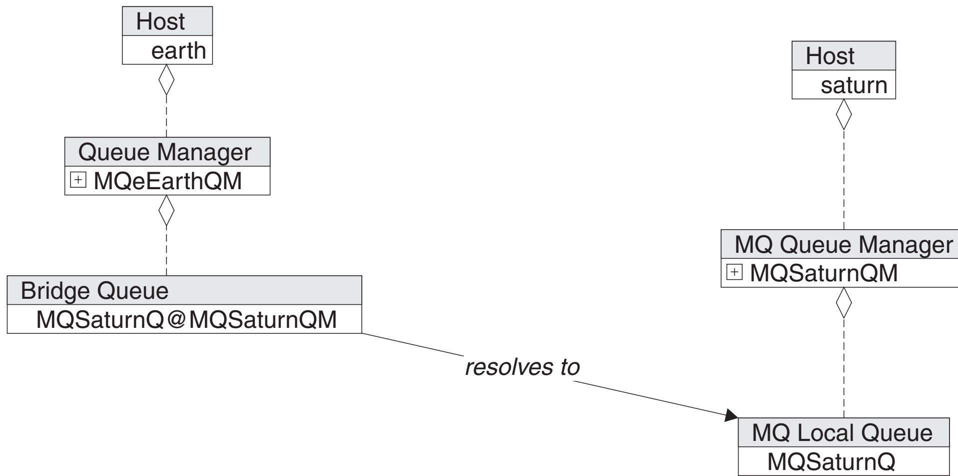


Figure 53. Pushing messages to MQ

Messages travel as expected across this remote queue definition, as shown below:

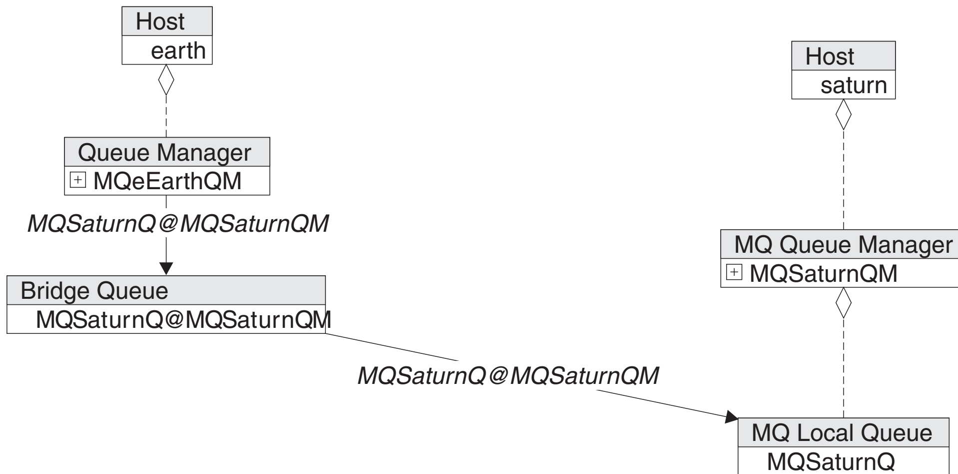


Figure 54. Messages travelling across a remote queue definition

This is exactly the same as a simple push message route between two queue managers, as shown below:

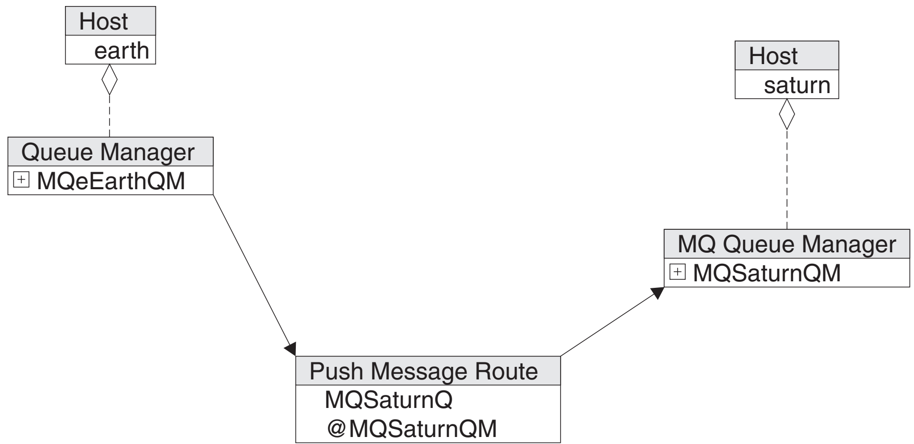


Figure 55. Simplified view of route pushing messages to MQ

Connecting a client to MQ via a bridge

A common topology is to allow messages to flow between MQ and a client MQe queue manager. This cannot happen directly, but requires an intermediate bridge-enabled MQeQueue manager. The client can then be a small footprint device with no knowledge of MQ. Additions are needed to allow a client (MQeMoonQM, on a device called moon) to communicate with MQ, as shown in the following diagram:

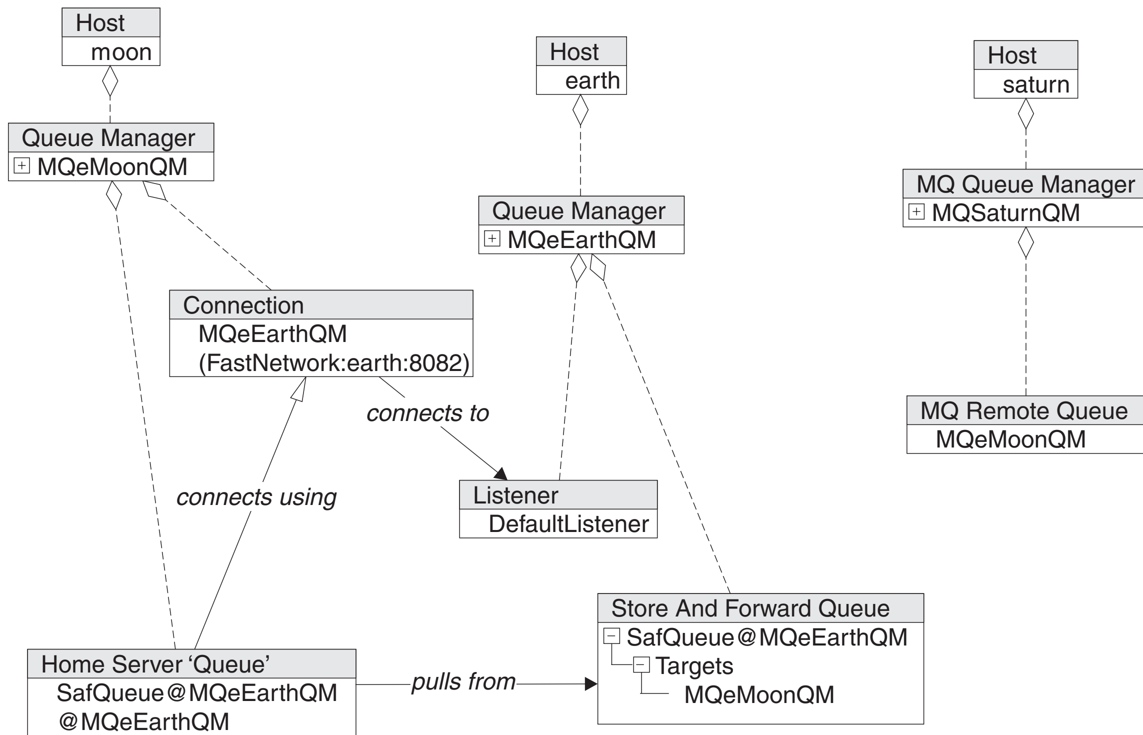


Figure 56. A client communicating with MQ

This adds the following:

- (Host)moon

- (QueueManager) MQeMoonQM on (Host)moon
- A connection definition from MQeMoonQM to a matching listener on MQeEarthQM to provide the connectivity between the two MQe queue managers.
- A store and forward queue on MQeEarthQM that accepts and holds messages for MQeMoonQM, and a home server queue on MQeMoonQM that pulls messages from the store and forward queue.
- A remote queue definition on the MQ queue manager that routes messages for MQeMoonQM to the transmission queue MQeEarth.XMITQ. This allows messages for MqeMoonQM to be placed on the transmission queue, from where they are pulled to MQeEarthQM.

The topology is more readily seen as message routes, as shown in the following diagram:

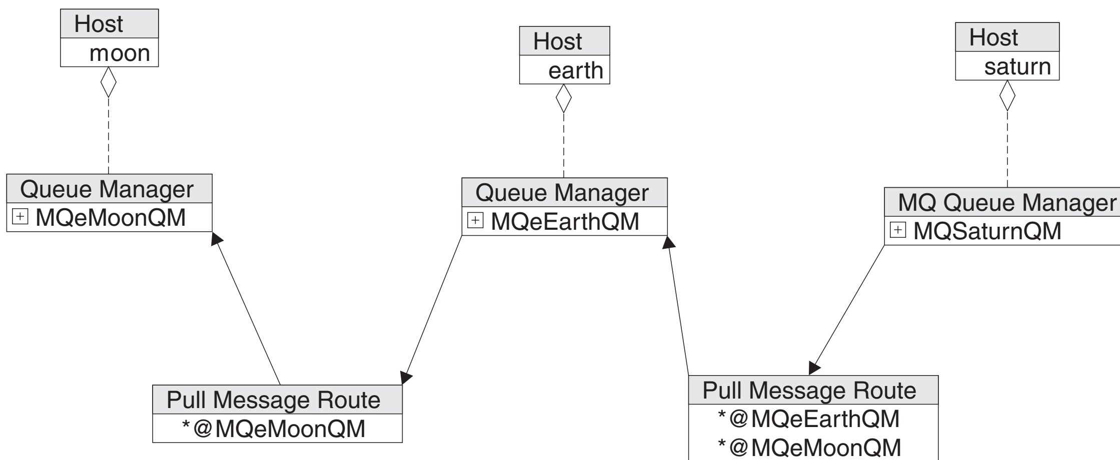


Figure 57. Simplified pull routes from MQ through an MQe gateway to an MQe device style queue manager

Messages can be pushed to MQ by using a via connection to chain remote queues, as shown below:

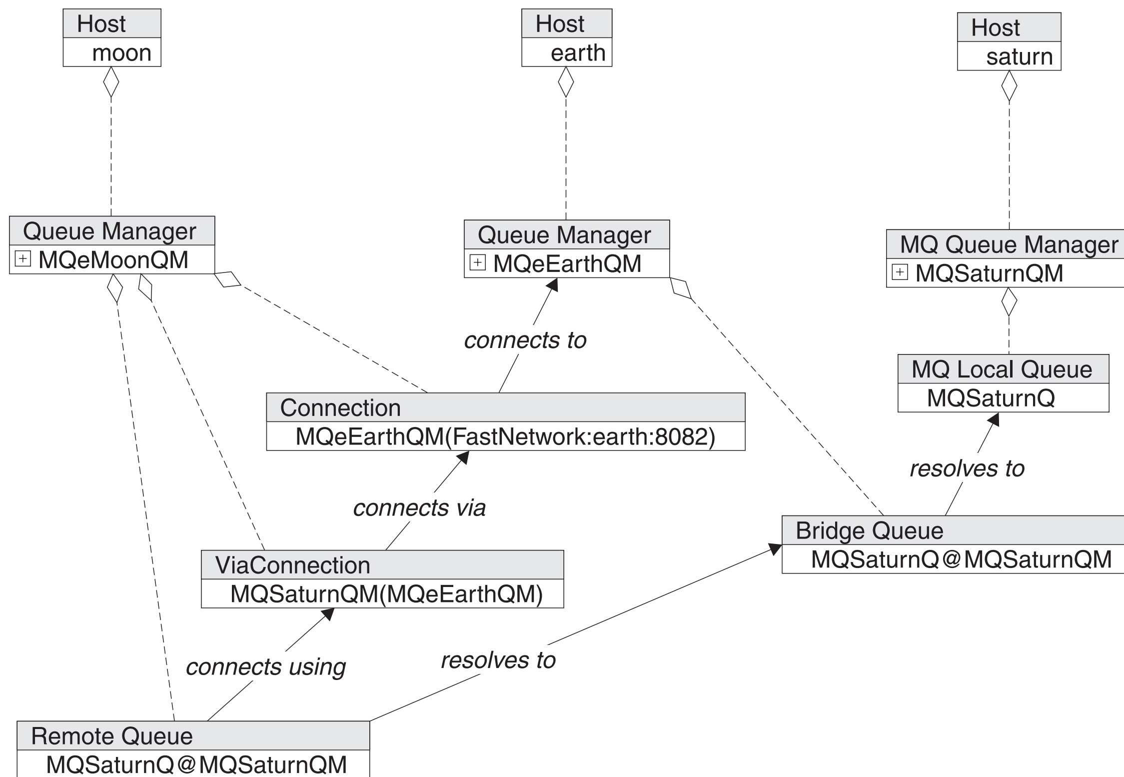


Figure 58. Pushing messages using a via connection

Here a via connection has been added to route messages destined for MQSaturnQM vian MQeEarthQM, and a remote queue definition for MQSaturnQ@MQSaturnQM has been added. The messages can now flow from the client to MQ, as shown in the following diagram:

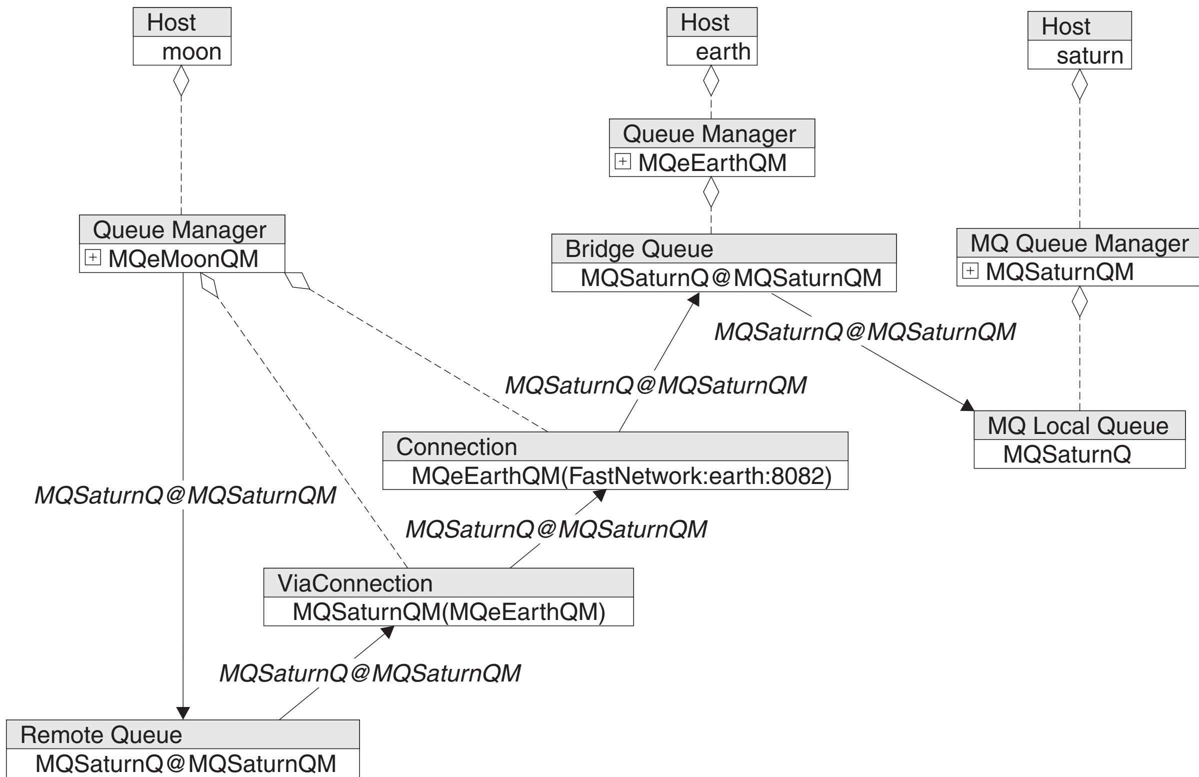


Figure 59. Pushing messages to MQ

This topology is more easily understood as a collection of message routes, as follows:

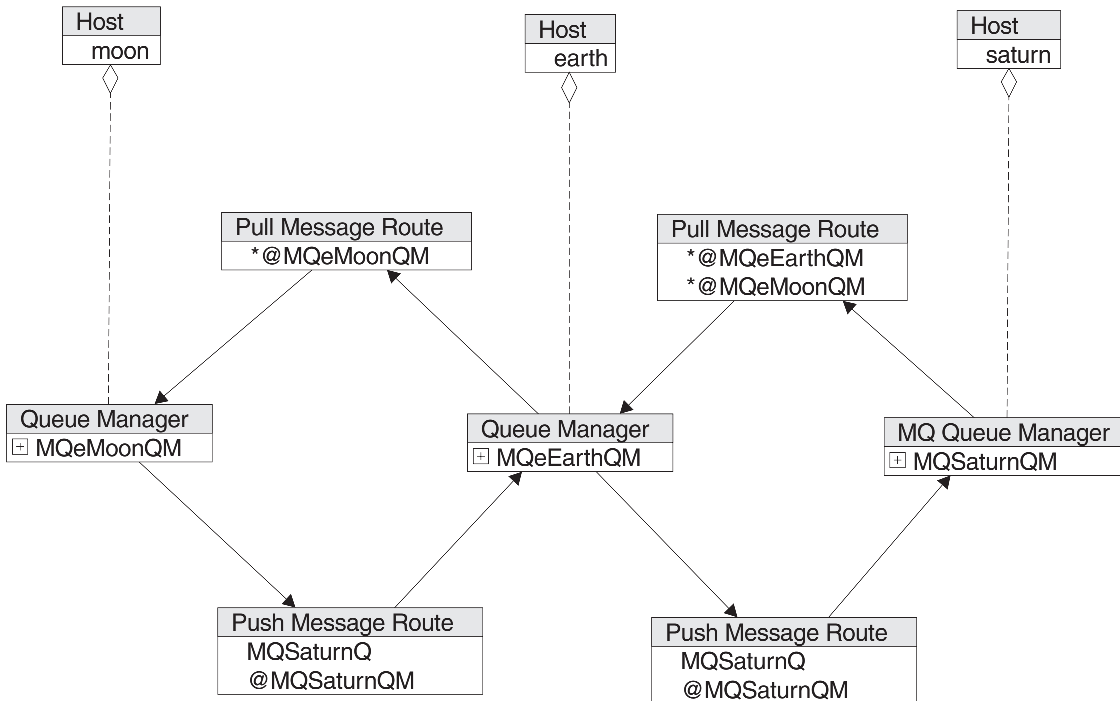


Figure 60. Simplified view showing routes which push messages from a device style MQe queue manager to an MQ queue manager

Pushing messages to MQ with a via connection

A common topology allows messages to flow between MQ and a client MQe queue manager. This cannot happen directly, but requires an intermediate bridge-enabled MQeQueue manager. The client can then be a small footprint device with no knowledge of MQ. If you start with the configuration we have above, the following additions are needed to allow a client (MQeMoonQM, on a device called moon) to communicate with MQ, as shown in the following diagram:

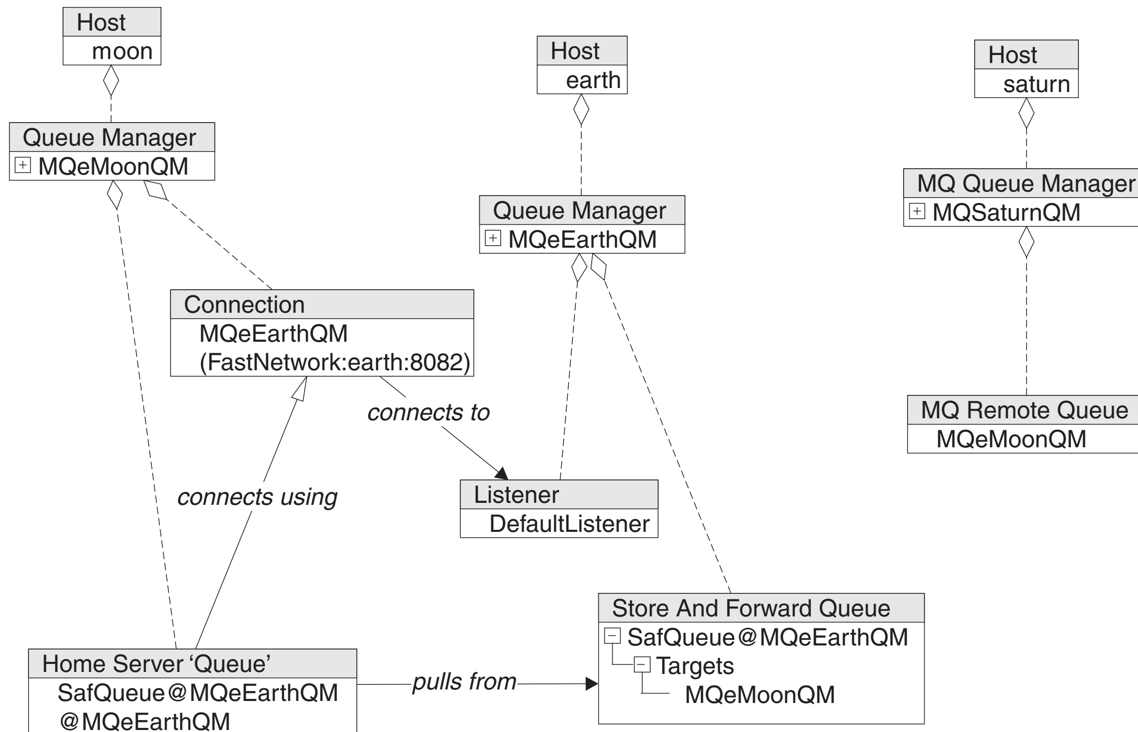


Figure 61. A client communicating with MQ

The following have been added:

- (Host)moon
- (QueueManager) MQeMoonQM on (Host)moon
- A connection definition from MQeMoonQM to a matching listener on MQeEarthQM to provide the connectivity between the two MQe queue managers.
- A store and forward queue on MQeEarthQM that accepts and holds messages for MQeMoonQM, and a home server queue on MQeMoonQM that pulls messages from the store and forward queue.
- A remote queue definition on the MQ queue manager that routes messages for MQeMoonQM to the transmission queue MQeEarth.XMITQ. This allows messages for MqeMoonQM to be placed on the transmission queue, from where they are pulled to MQeEarthQM.

The topology is more readily seen as message routes, as shown in the following diagram:

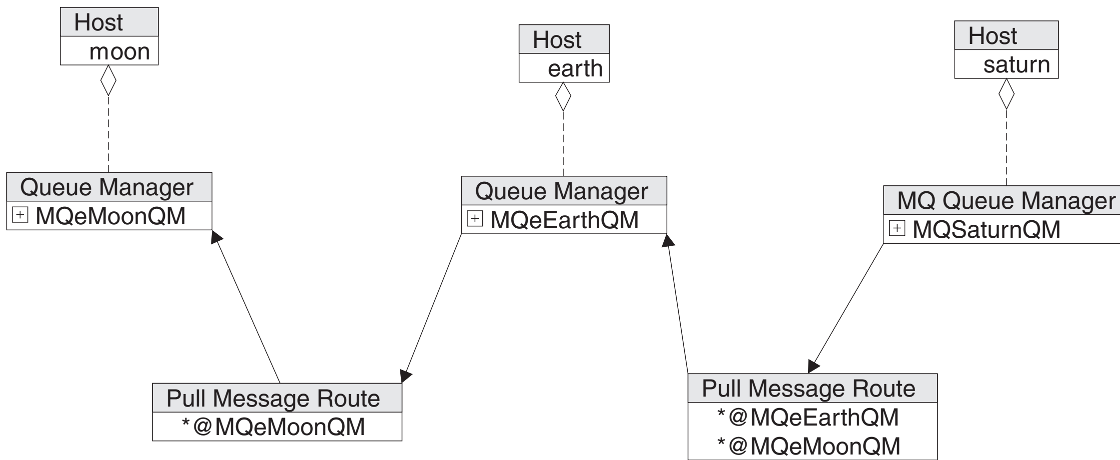


Figure 62. Simplified pull routes from MQ through an MQe gateway to an MQe device style queue manager

Messages can be pushed to MQ by using a via connection to chain remote queues, as shown in the following diagram:

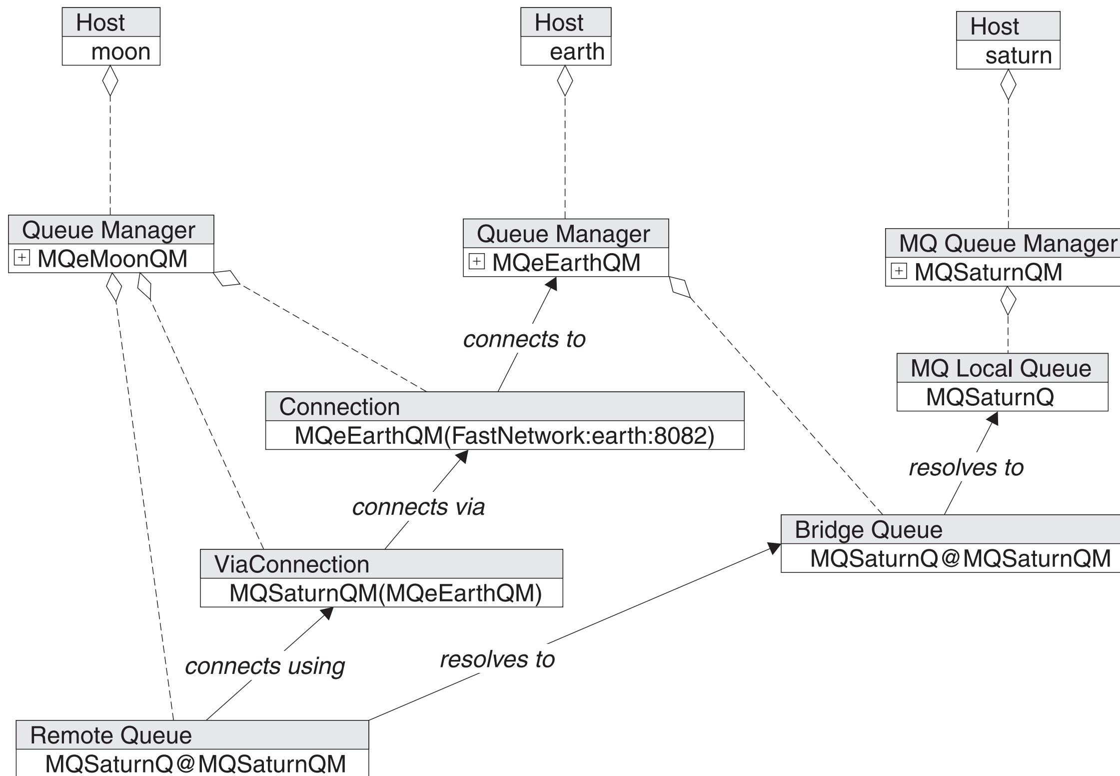


Figure 63. Pushing messages using a via connection

Here we have added a via connection, to route messages destined for MQSaturnQM vian MQeEarthQM, and we have added a remote queue definition for MQSaturnQ@MQSaturnQM. The messages can now flow from the client to MQ, as shown in the following diagram:

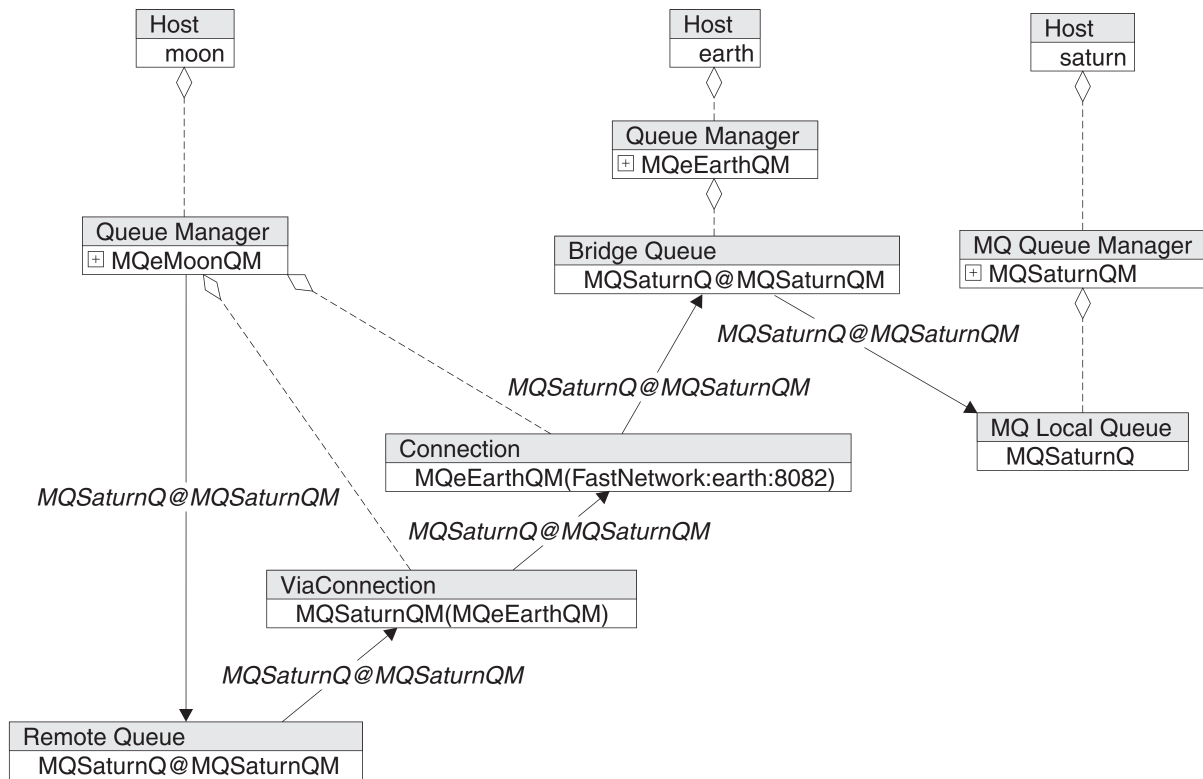


Figure 64. Pushing messages to MQ

This topology is more easily understood as a collection of message routes, as shown in the following diagram:

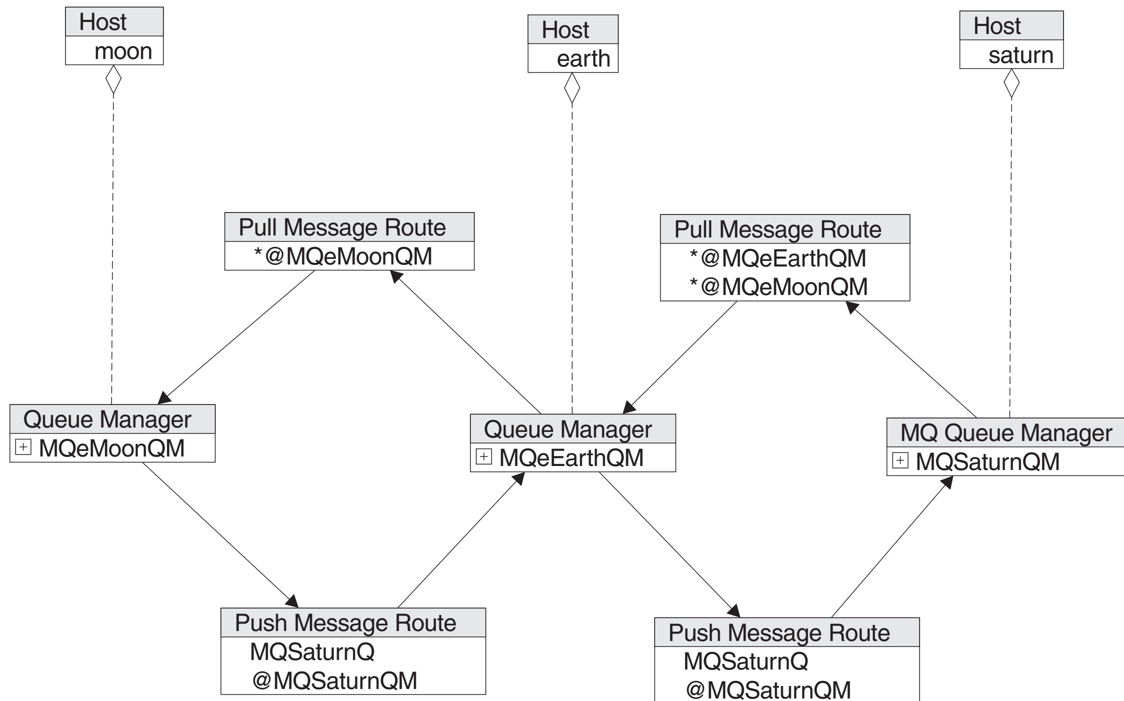


Figure 65. Simplified view showing routes which push messages from a device style MQe queue manager to an MQ queue manager

Security considerations

Remote queue definitions define the security requirements that must be satisfied by channels moving messages to target queues. The queue manager attribute rule defines the rules for upgrading channels; consequently with a sufficiently flexible rule, multiple security requirements can be met by a single channel.

When a message must be stored on a queue, either en route or at the destination, then the queue attribute rule determines if the channel security meets the requirements of the queue. Note however that there are message transfers that do not involve a channel, for example, when a home server places a message it has received from a store queue on to its destination queue. In these cases there are no security requirements to be satisfied in the transfer, but the message will be stored in its destination queue in a manner controlled by that queue's security characteristics. When the home server queue gets the message from the store queue, a channel is involved (with characteristics determined by the home server queue and which must be acceptable to the store queue). However, when the home server queue passes the message to the destination queue, there are no channel characteristics to be compared with the destination queue's security characteristics.

In a single hop, message transfer, the security checking is between the source and target queue managers. In multiple hop, asynchronous message transfers, security checking occurs stepwise over each hop.

Resolution rules

Resolution rules always start with a message being presented to a queue manager, with a specified destination queue manager name and a specified destination queue name. This is equivalent to the API call putMessage(queueManagerName, queueName, msg,....). The destinationQueueManagerName and destinationQueueName must identify a local queue onto which the message should eventually be placed.

Rule 1: Resolve queue manager aliases

If the queue manager has an alias mapping destinationQueueManagerName to another name, for example realQueueManagerName, then this substitution is made first, and the call:

```
putMessage(destinationQueueManagerName, destinationQueueName
```

is effectively transformed to

```
putMessage(realQueueManagerName, destinationQueueName.
```

From this point on destinationQueueManagerName is completely forgotten, and realQueueManagerName is used.

Queue resolution

The queue manager now looks for a queue to place the message on, selecting the queue with the best match according to the rules shown in *Exact match*, *Queue alias match*, *S&F queue*, *Queue discovery*, and *Failure*, below:

’Exact’ match

Local queue or remote queue definition where the queue name matches the destinationQueueName and the queue’s queue manager name matches the destinationQueueManagerName.

The term ’queues queue manager name’ needs to be explained further. For a local queue this is the same as the name of the queue manager where the queue resides. For a local queue localQ@localQM, localQM is the queue’s queue manager name.

For a remote queue definition remoteQ@remoteQM residing on localQM, the queues queue manager name is remoteQM.

Queue Alias Match

If a queue (remote definition or local) has a matching queue manager name and an alias and this alias matches destinationQueueName then this queue will considered a match. Effectively the put message call:

```
putMessage(destinationQueueManagerName, queueAliasName
```

is transformed to

```
putMessage(destinationQueueManagerName, realQueueName.
```

at this point. The original name of the queue used in the put call is entirely forgotten from this point on in the resolution.

S&F queue

If there is no exact match the queue manager searches for an inexact match. An inexact math is a Store and Forward queue that will accept messages for the given queue manager name. The search for a store and forward queue ignores the destinationQueueName. If an appropriate Store And Forward queue is found, then the message is put to it, using the destinationQueueManagerName and destinationQueueName, and the StoreAndForward queue stores the destination with the message.

Queue Discovery

If no queue has been found that will accept the message, and the message is not for a local queue, the queue manager tries to find the remote destination queue and create a remote queue definition for it automatically. This is called queue discovery. The queue manager can only perform discovery if:
- There is a connection definition to the destination queue manager
- There is an active communications path to the destination queue manager
- The destination queue exists
- There is a via connection to a queue manager where a remote connection definition exists

If discovery is successful, the newly created remote queue definition is used. This behaves as if an exact match on a remote queue definition had been found in the first place.

The remote queue definition created by discovery is always synchronous, even if the queue to which it resolves is asynchronous, or is a Store and forward queue.

Failure

If no queue has been found by the above steps, the message put is deemed to have failed.

Push across network

A message placed upon a remote queue is pushed across the network. The queue first locates a connection definition with the correct name, and then puts the message to the remote queue manager using the connection definition as the entry to the communications link.

The queue seeks a connection definition whose name is the same as the queue's queue manager name. The connection may be a normal connection, or a via connection.

Normal

A normal connection points to a listener upon the destination queue manager. The put message command is routed directly to the destination queue manager. The putMessage call is then resolved just as if it had been placed on the queue manager via the API.

Via

A via connection points at another connection called the 'real' connection. All commands performed on the via connection are delegated to the real connection. Via connections can be chained, and so the command may travel 'via' several indirections before reach a real connection. The names of the put message destination are not changed by the use of a via connection.

Eventually the command is routed to a 'normal' connection definition, then across the network to a queue manager, where the message put is resolved.

Home server pulling

Home server queues pull messages from Store and forward queues. The route of the pull spans only a single network hop. Only messages for the queue manager hosting the home server queue are pulled down. Messages pulled from the store and forward queue are presented to the queue manager using a normal put method call, and are then resolved as normal. The messages pulled down this way should all be destined for local queues.

Using aliases

Introduction to the use of aliases with MQe queues and queue managers

Aliases can be assigned for MQe queues to provide a level of indirection between the application and the real queues. For example, a queue can be given a number of aliases and messages sent to any of these names will be accepted by the queue.

Using queue aliases

This topic describes the ways in which aliasing can be used with MQe queues.

Merging applications

Queue aliasing can be used to merge applications. For example, imagine you have the following configuration:

- A client application that puts data to queue Q1
- A server application that takes data from Q1 for processing
- A client application that puts data to queue Q2
- A server application which takes data from Q2 for processing

Some time later the two server applications are merged into one application supporting requests from both the client applications. It may now be appropriate for the two queues to be changed to one queue. For example, you may delete Q2, and add an alias of the Q1 queue, calling it Q2. Messages from the client application that previously used Q2 are automatically sent to Q1.

Upgrading applications

Queue aliasing can be used to upgrade applications. For example, imagine you have the following configuration:

- A queue Q1
- An application that gets messages from Q1
- An application that puts messages to Q1

You then develop a new version of the application that gets the messages. You can make the new application work with a queue called Q2. You can define a queue called Q2 and use it to exercise the new application. When you want it to go live, you let the old version clear all traffic off the Q1 queue, and then create an alias of Q2 called Q1. The application that puts to Q1 will still work, but the messages will end up on Q2.

Using different transfer modes to a single queue

Suppose you have a queue MY_Q_ASYNC on queue manager MQE1. Messages are passed to MY_Q_ASYNC by a different queue manager MQE2, using a remote queue definition that is defined as an asynchronous queue. Now suppose your application periodically wants to get messages in a synchronous manner from the MY_Q_ASYNC queue.

The recommended way to achieve this is to add an alias to the MY_Q_ASYNC queue, perhaps called MY_Q_SYNC. Then define a remote queue definition on your MQE2 queue manager, that references the MY_Q_SYNC queue. This provides you with two remote queue definitions. If you use the MY_Q_ASYNC definition, the messages are transported asynchronously. If you use the MY_Q_SYNC definition, synchronous message transfer is used.

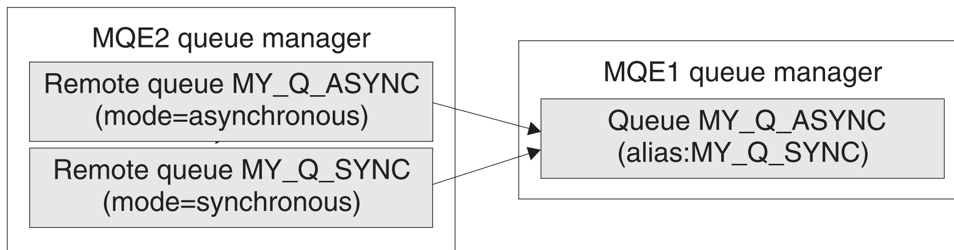


Figure 66. Two modes of transfer to a single queue

Using queue manager aliases

This topic describes the ways in which aliasing can be used with MQe queue managers.

Addressing a queue manager with several different names

Suppose you have a queue manager `SERVER23QM` on the server `SAMPLEHOST`, listening on port `8082`. You have an application `SERVICEX` that accesses this queue manager, and wants to refer to the queue manager as `SERVICEXQM`. This can be achieved using an alias for the queue manager as follows:

- **Configure a connection on the SERVER23QM :**

  Connection Name/Target queue manager:
  : `SERVICEXQM`

  Description:
  : Alias definition to enable `SERVER23QM` to receive messages sent to `SERVICEXQM`

  Channel:
  : "null"

  Network Adapter:
  : "null"

  Network adapter options:
  : "null"

- **Create a local queue on the `SERVER23QM` queue manager:**

  Queue Name:
  : `SERVICEXQ`

  Queue Manager:
  : `SERVER23QM`

  The server-side application takes messages from this queue, and process them, sending messages back to the client.

an MQe application can now put messages to the SERVICEXQ on either the SERVER23QM queue manager, or the SERVICEXQM queue manager. In either case, the message will arrive on the SERVICEXQ.

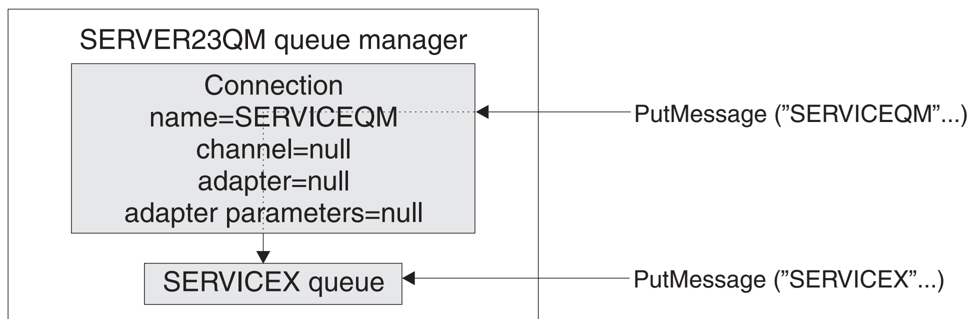


Figure 67. Addressing a queue manager with two different names

If the SERVICEXQ queue is moved to another queue manager, the connection alias can be set up on the new queue manager, and the applications do not need to be changed.

Different routings from one queue manager to another

Using the scenario in "Addressing a queue manager with several different names" on page 134, an MQe queue manager on a mobile device (MOBILE0058QM) can now access the SERVICEXQ queue in a number of different ways.

Aliasing on the sending side:

Using this method of routing, the receiving queue manager does not know that the sending queue manager has given it an alias name. The aliasing is confined to the sending queue manager only.

On the mobile device:

- Create a connection from MOBILE0058QM to the SERVER23QM queue manager:

  Connection name
    SERVER23QM

  Network Adapter parameter
    Network:SAMPLEHOST:8082
- Create an alias called SERVICEXQM for queue manager SERVER23QM

When a message is sent from the mobile device application to the SERVICEXQM queue manager, MQe maps the SERVICEXQM name to SERVER23QM in the connection , and sends the message to the SERVER23QM queue manager.

If the Mobile58QM then wished to send its messages to a different server queue manager, Server24QM, it would remove the alias SERVICEXQM from the Server23QM connection, and add it to a Server24QM connection. This has no impact on the receiving queue managers, or the sending applications.

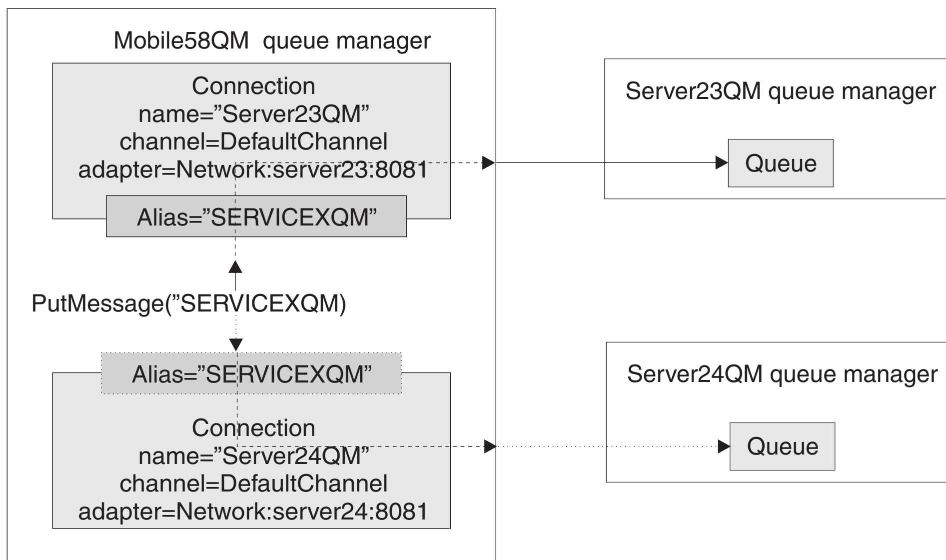


Figure 68. Addressing a queue manager with two different names

Virtual queue manager on the receiving side:

Using this method, the sending queue managers think that their messages are routed through an intermediate queue manager before reaching the target queue manager. The target queue manager doesn't actually exist. The 'intermediate' queue manager captures all the message traffic for this virtual target queue manager.

On the mobile device:

- Create a connection from `MOBILE0058QM` to the `SERVER23QM` queue manager:

  Connection name
  : `SERVER23QM`

  Network Adapter parameter
  : `Network:SAMPLEHOST:8082`

- Create a second connection to the `SERVICEXQM` that routes messages through the first connection:

  Connection name
  : `SERVICEXQM`

  Network Adapter parameter
  : `SERVER23QM`

  Note: This is not an alias. It is a *via routing*, indicating that messages headed for `SERVICEXQM` are to be routed via the `SERVER23QM` queue manager on the receiving side.

The via routing on the mobile device causes any messages that are put to `SERVICEXQM` to be directed to `Server23QM`. `Server23QM` gets the messages and notes that they are destined for the `SERVICEXQM` queue manager. It resolves the `SERVICEXQM` name and finds that it is an alias which represents the `Server23QM`

queue manager (itself). The `Server23QM` queue manager then accepts the messages and puts them onto the queue.

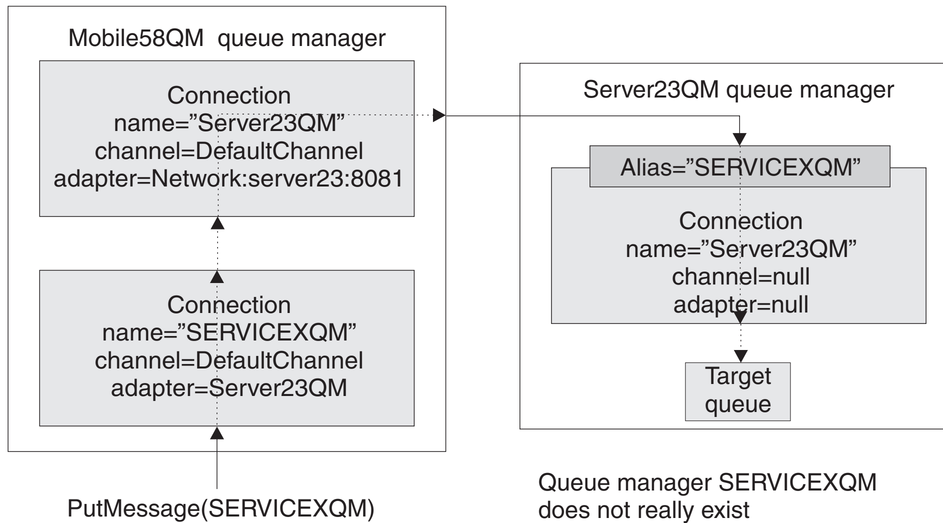


Figure 69. Addressing a queue manager with two different names

As an alternative to the above, you can keep the `SERVICEXQM` in existence, but move it from its original machine to the same machine (but a different JVM) as the `Server23QM` queue manager. `SERVICEXQM` needs to listen on a different port, so the connection from `Server23QM` to `SERVICEXQM` needs to be changed as well.

Using adapters

Describes the use of storage adapters and communications adapters in MQe applications, and explains how to write your own adapters

This chapter describes how to implement adapters in an MQe application. You can use MQe adapters to map MQe to storage or communications device interfaces. You can also write your own adapters.

This chapter contains the following sections:
- Storage adapters
- Communications adapters
- How to write adapters

Storage adapters

MQe provides the following storage adapters:

Storage adapters

MQeCaseInsensitiveDiskAdapter
Provides support for case insensitive matching when locating a specific file in permanent storage.

MQeDiskFieldsAdapter
Provides support for reading and writing to persistent storage.

MQeMappingAdapter
Provides support for mapping long file names to short file names.

MQeMemoryFieldsAdapter
Provides support for reading and writing to non-persistent storage.

MQeMidpFieldsAdapter
Provides support for reading and writing to permanent storage within a MIDP environment.

MQeReducedDiskFieldsAdapter
Provides support for high speed writing to permanent storage.

Note that you cannot alter the behavior of these adapters. For more information on the specific behavior of each storage adapter, refer to the MQe Java API Programming Reference and the MQe C API Programming Reference.

Communications adapters

MQe provides the following communications adapters:.

Communications adapters

MQeMidpHttpAdapter
Provides support for reading and writing to the network using the HTTP 1.0 protocol in a MIDP environment.

MQeTcpipHistoryAdapter
Provides support for reading and writing to the network using the TCP protocol. This adapter provides the best TCP performance by chaching recently used data. Therefore, we recommend that you use this adapter.

MQeTcpipLengthAdapter
Provides support for reading and writing to the network using the TCP protocol.

MQeTcpipHttpAdapter
Provides support for reading and writing to the network using the HTTP 1.0 protocol. Also provides support for passing HTTP requests through proxy servers.

Note: If using the Microsoft JVM, the http:proxyHost and http:proxyPort properties are automatically set by the JVM using the settings in the Internet Explorer. If the use of proxies is not required for MQe, set the http.proxySet Java property to false.

MQeUdpipBasicAdapter
Provides support for reading and writing to the network using the UDP protocol. This adapter uses only one port on the server. The behavior of this adapter is particularly sensitive to the various Java property settings, as detailed in the MQe Java Programming Reference.

MQeWESAuthenticationAdapter
Provides support for passing HTTP requests through MQe authentication proxy servers and transparent proxy servers.

You can modify the behavior of these adapters using Java properties. For more information on how to use these properties and their effect on each communications adapter, refer to the MQe Java API Programming Reference.

You can also write your own adapters to tailor MQe for your own environment. The next section describes some adapter examples that are supplied to help you with this task.

How to write adapters

You can also write your own adapters to tailor MQe for your own environment. This topic describes some adapter examples that are supplied to help you with this task.

This example is not intended as a replacement for the adapters that are supplied with MQe, but as a simple introduction on how to create a communications adapter.

To use your communications adapter, you must specify the correct class name when creating the listener on the server queue manager, and specify the connection definition on the client queue manager.

All communications adapters must inherit from MQeCommunicationsAdapter and must implement the required methods. In order to show how this might be done we shall use the example adapter, `examples.adapters.MQeTcpipLengthGUIAdapter`. This is a simple example that accepts data to be written. It also places the data length and the amount of data to be written to standard out, at the front of the data. When the adapter reads data, the data length is written to standard out. Proper error checking and recovery is not carried out. This must be added to any adapter written by a user.

MQe adapters use the default constructor. For this reason, an `activate()` method is used in order to set up the adapter with an `open()` method used to prepare the adapter for communication.

The `activate()` method is called only once in the life-cycle of an adapter and is, therefore, used to set up the information from MQePropertyProvider. The MQePropertyProvider looks internally to verify that the specified property is available. If it is not available, it checks the Java properties. In this way, it is possible for a user to specify a property that may be set by the application or JVM command line. The MQeCommunicationsAdapter provides two variables that allow the adapter to identify its role within the communications conversation:

- If the adapter is being used by the MQeListener, the variable `listeningAdapter` is set to true.
- If the adapter has been created by the listening adapter in response to an incoming request, the `responderAdapter` variable is set to true.

The following code, taken from the `activate()` method, shows how to obtain the information from the MQePropertyProvider.

```
if (!listeningAdapter) {
  // if we are not a listening adapter we need the
  address of the server
  address = info.getProperty
      (MQeCommunicationsAdapter.COMMS_ADAPTER_ADDRESS);
}
```

The `open()` method is called before each conversation and must, therefore, be used to set information that needs to be reset for each request or response. For example, an adapter that is not persistent needs to create a socket each time it is opened. The following code shows the use of the variables that identify the role of the adapter role within the conversation:

```
    if (listeningAdapter && null == serverSocket) {
    serverSocket = new ServerSocket(port);
  } else if (!responderAdapter && null == mySocket) {
    mySocket = new Socket(InetAddress.getByName(address), port);
  }
```

Once the `activate()` and `open()` methods have been called, the listening adapter `waitForContact` method is called. This method must wait at named location. In an IP network, this will be a named port. When a request is received, a new adapter is created.

Note: This method must set the listeningAdapter to false and the responderAdapter to true.

Once the adapter has been set up correctly, you must must returned it to the caller. The following code shows how to do this:

```
MQeTcpipLengthGUIAdapter clientAdapter =
    (MQeTcpipLengthGUIAdapter)
      MQeCommunicationsAdapter.createNewAdapter(info);

    // set the boolean variables so the adapter
  //  knows it is a responder. the listening
    // variable will have been set to true as
  //  the MQePropertyProvider has the relevant
    // information to create
  //  this listening adapter.  We must therefore reset the
    // listeningAdapter variable to false and the
  //responderAdapter variable to true.
  clientAdapter.responderAdapter = true;
  clientAdapter.listeningAdapter = false;

  // Assign the new socket to this new adapter
  clientAdapter.setSocket(clientSocket);
  return clientAdapter;
```

The initiator adapter and responder adapter are responsible for the main part of the conversation. The initiator starts the conversation. The responder is created by the listening adapter, reads the request that is passed back to MQe, which then writes a response. The adapter determines how the read and the write are undertaken. The example uses a BufferedInputStream and a BufferedOutputStream.

Note: Use a a non-blocking mode of reading and writing. This enables the adapter to respond to requests to shutdown.

The following code, taken from the `waitForContact()` method, shows how the non-blocking read can be written. As MQe supports all Java runtime environments we are unable to use Java version 1.4 specific classes for our examples, although this version does contain new non-blocking classes

```
do {
      try {
        clientSocket = serverSocket.accept();
      } catch (InterruptedIOException iioe) {
          if (MQeThread.getDemandStop()) {
             throw iioe;
          }
      }
    } while (null == clientSocket);
```

An example communications adapter

This example uses the standard Java classes to manipulate TCPIP and adds a protocol of its own on top. This protocol has a header consisting of a four byte length of the data in the data packet followed by the actual data. This is so that the receiving end knows how much data to expect.

This example is not meant as a replacement for the adapters that are supplied with MQe but rather as a simple introduction into how to create communications adapters. In reality, much more care should be taken with error handling, recovery, and parameter checking. Depending on the MQe configuration used, the supplied adapters may be sufficient.

A new class file is constructed, inheriting from MQeAdapter. Some variables are defined to hold this adapter's instance information, that is the name of the host, port number and the output stream objects.

Note: With communications, ensure that the connection information is correct. For example, the http connection in J2ME has no timeout implementation. In J2SE, the client times out with an IO Exception. In Midp the server times out. If the default read-timeout has been increased for the J2SE client, the same exception is thrown, that is `com.ibm.mqe.MQeException: Data: (code=7)`. This is because the server writes back the exception to the client and the client cannot restore this data.

The MQeAdapter constructor is used for the object, so no additional code needs to be added for the constructor.

```
public class MyTcpipAdapter extends MQeAdapter
  {
  protected     String               host         = "";
  protected     int                  port         = 80;
  protected     Object               readLock     = new Object( );
  protected     ServerSocket         serversocket = null;
  protected     Socket               socket       = null;
  protected     BufferedInputStream  stream_in    = null;
  protected     BufferedOutputStream stream_out   = null;
  protected     Object               writeLock    = new Object( );
```

Next the `activate` method is coded. This is the method that extracts from the file descriptor the name of the target network address if a connector, or the listening port if a listener. The fileDesc parameter contains the adapter class name or alias name, and any network address data for the adapter for example `MyTcpipAdapter:127.0.0.1:80`. The thisParam parameter contains any parameter data that was set when the connection was defined by administration, the normal value would be "?Channel". The thisOpt parameter contains the adapter setup options that were set by administration, for example `MQe_Adapter_LISTEN` if this adapter is to listen for incoming connections.

```
  public void   activate( String   fileDesc,
                          Object   thisParam,
                          Object   thisOpt,
                          int      thisValue1,
                          int      thisValue2 ) throws Exception
    {
    super.activate( fileDesc,
                    thisParam,
                    thisOpt,
                    thisValue1,
                    thisValue2 );
    /* isolate the TCP/IP address -
```

```
                "MyTcpipAdapter:127.0.0.1:80"     */
  host = fileId.substring( fileId.indexOf( ':' ) + 1 );
  i    = host.indexOf( ':' );
  /* find delimiter     */
  if ( i > -1 )
  /* find it ?          */
    {
    port = (new Integer( host.substring( i + 1 ) )).intValue( );
    host = host.substring( 0, i );
    }
  }
```

The `close` method needs to be defined to close the output streams and flush any remaining data from the stream buffers. `Close` is called many time during a session between a client and a server, however, when the channel has completely finished with the adapter it calls MQe with the option `MQe_Adapter_FINAL`. If the adapter is to have one socket connection for the life of the channel then the call with `MQe_Adapter_FINAL` set, is the one to use to actually close the socket, other calls should just flush the buffers. If however a new socket is to be used on each request, then each call to MQe should close the socket, subsequent `open` calls should allocate a new socket:

```
public void    close(  Object  opt ) throws Exception
  {
  if ( stream_out   != null )
/* output stream ?     */
    {
    stream_out.flush();
/* empty the  buffers  */
    stream_out.close();
/* close it            */
    stream_out = null;
/* clear               */
    }
  if ( stream_in    != null )
/* input stream ?      */
    {
    stream_in.close();
/* close it            */
    stream_in = null;
/* clear               */
    }
  if ( socket       != null )
/* socket ?            */
    {
    socket.close();
/* close it            */
    socket = null;
/* clear               */
    }
  if ( serversocket != null )
/* serversocket ?      */
    {
    serversocket.close();
/* close it            */
    serversocket = null;
/* clear               */
    }
  host = "";
  port = 80;
  }
```

The `control` method needs to be coded to handle an `MQe_Adapter_ACCEPT` request, to accept an incoming connect request. This is only allowed if the socket is a listener (a server socket). Any options that were specified for the listen socket

(excluding MQe_Adapter_LISTEN) are copied to the socket created as a result of the accept. This is accomplished by the use of another control option MQe_Adapter_SETSOCKET this allows a socket object to be passed to the adapter that was just instantiated.

```
public Object control( Object opt, Object ctrlObj ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_LISTEN      ) &&
         checkOption( opt, MQe.MQe_Adapter_ACCEPT ) )
      {
      /* CtrlObj - is a string representing the
              file descriptor of the */
      /*          MQeAdapter object to be returned e.g. "MyTcpip:"   */
      Socket  ClientSocket = serversocket.accept();
    /* wait connect  */
      String  Destination  = (String) ctrlObj;
    /* re-type object*/
      int i  = Destination.indexOf( ':' );
      if ( i < 0 )
        throw new MQeException( MQe.Except_Syntax,
                                "Syntax:" + Destination );
      /* remove the Listen option   */
      String NewOpt = (String) options;
    /* re-type to string   */
      int j  = NewOpt.indexOf( MQe.MQe_Adapter_LISTEN );
      NewOpt = NewOpt.substring( 0, j ) +
           NewOpt.substring
            ( j + MQe.MQe_Adapter_LISTEN.length( ) );
                MQeAdapter Adapter = MQe.newAdapter
                    ( Destination.substring( 0,i+1 ),
                                 parameter,
                                 NewOpt + MQe_Adapter_ACCEPT,
                                 -1,
                                 -1 );
      /* assign the new socket to this new adapater  */
      Adapter.control( MQe.MQe_Adapter_SETSOCKET, ClientSocket);
      return( Adapter );
      }
    else
    if ( checkOption( opt, MQe.MQe_Adapter_SETSOCKET ) )
      {
      if ( stream_out != null )  stream_out.close();
      if ( stream_in  != null )  stream_in .close();
      if ( ctrlObj    != null )
    /* socket supplied ?*/
        {
        socket     = (Socket) ctrlObj;
    /* save the socket     */
        stream_in  = new BufferedInputStream (socket.getInputStream ());
        stream_out = new BufferedOutputStream(socket.getOutputStream());
        }
    else
      return( super.control( opt, ctrlObj ) );
  }
```

The open method needs to check for a listening socket or a connector socket and create the appropriate socket object. Reinitialization of the input and output streams is achieved by using the control method, passing it a new socket object. The opt parameter may be set to MQe_Adapter_RESET, this means that any previous operations are now complete any new reads or writes constitute a new request.

```
  public void  open( Object opt ) throws Exception
    {
    if ( checkOption( MQe.MQe_Adapter_LISTEN ) )
      serversocket = new ServerSocket( port, 32 );
```

```
    else
      control( MQe.MQe_Adapter_SETSOCKET,
          new Socket( host, port ) );
    }
```

The read method can take a parameter specifying the maximum record size to be read.

This example calls internal routines to read the data bytes and do error recovery (if appropriate) then return the correct length byte array for the number of bytes read. Ensure that only one read at a time occurs on this socket. The opt parameter may be set to:

MQe_Adapter_CONTENT
read any message content

MQe_Adapter_HEADER
read any header information

```
{ public byte[] read( Object opt, int recordSize ) throws Exception

    int Count = 0;
  /* number bytes read   */
    synchronized ( readLock )
  /* only one at a time  */
      {
      if ( checkOption(opt, MQe.MQe_Adapter_HEADER )  )
        {
        byte lreclBytes[] = new byte[4];
  /* for the data length */
        readBytes( lreclBytes, 0, 4 );
  /* read the length     */
        int  recordSize = byteToInt( lreclBytes, 0, 4 );
        }
      if ( checkOption( opt, MQe.MQe_Adapter_CONTENT   ) )
        {
        byte Temp[] = new byte[recordSize];
  /* allocate work array */
        Count = readBytes( Temp, 0, recordSize);/* read data   */
        }
      }
    if ( Count < Temp.length )
  /* read all length ?   */
      Temp    = MQe.sliceByteArray( Temp, 0, Count );
    return ( Temp );
  /* Return the data     */
    }
```

The readByte method is an internal routine designed to read a single byte of data from the socket and to attempt to retry any errors a specific number of times, or throw an end of file exception if there is no more data to be read.

```
  protected int readByte( ) throws Exception
    {
    int intChar    = -1;
  /* input characater    */
    int RetryValue =  3;
  /* error retry count   */
    int Retry = RetryValue + 1;
  /* reset retry count   */
    do{
  /* possible retry      */
      try
  /* catch io errors     */
        {
        intChar = stream_in.read();
```

```
/* read a character    */
      Retry   = 0;
 /* dont retry          */
      }
    catch ( IOException e )
/* IO error occured    */
      {
      Retry   = Retry - 1;
/* decrement           */
      if ( Retry == 0 )  throw e;
/* more attempts ?     */
      }
    } while ( Retry != 0 );
/* more attempts ?     */
  if ( intChar == -1 )
/* end of file ?       */
    throw new EOFException();
/* ... yes, EOF        */
  return( intChar );
/* return the byte     */
  }
```

The readBytes method is an internal routine designed to read a number of bytes of data from the socket and to attempt to retry any errors a specific number of times, or throw an end of file exception if there is no more data to be read.

```
protected int readBytes( byte buffer[],
        int offset, int recordSize )
  throws Exception
  {
  int RetryValue = 3;
  int i = 0;
/* start index         */
  while ( i < recordSize )
/* got it all in yet ? */
    {
/* ... no              */
    int NumBytes = 0;
/* read count          */
    /* retry any errors based on the QoS Retry value */
    int Retry = RetryValue + 1;
/* error retry count   */
    do{
/* possible retry      */
      try
/* catch io errors     */
        {
        NumBytes = stream_in.read( buffer,
        offset + i, recordSize - i );
        Retry    = 0;
/* no retry            */
        }
      catch ( IOException e )
/* IO error occured    */
        {
        Retry    = Retry - 1;
/* decrement           */
        if ( Retry == 0 )  throw e;
/* more attempts ?     */
        }
      } while ( Retry != 0 );
/* more attempts ?     */
    /* check for possible end of file  */
    if ( NumBytes < 0 )
/* errors ?            */
      throw new EOFException( );
/* ... yes             */
```

```
      i = i + NumBytes;
  /* accumulate         */
      }    return ( i );
  /* Return the count    */
    }
```

The `readln` method reads a string of bytes terminated by a 0x0A character it will ignore 0x0D characters.

```
    {
    synchronized ( readLock )
 /* only one at a time  */
      {
      /* ignore the 4 byte length   */
      byte lreclBytes[] = new byte[4];  /* for the data length */
      readBytes( lreclBytes, 0, 4 );
/* read the length     */

      int intChar    = -1;
 /* input characater    */
      StringBuffer Result = new StringBuffer( 256 );
      /* read Header from input stream          */
      while ( true )
 /* until "newline"     */
        {
        intChar = readByte( );
/* read a single byte  */
        switch ( intChar )
/* what character      */
          {
          case -1:
/* ... no character    */
            throw new EOFException();
/* ... yes, EOF        */
          case 10:
/* eod of line         */
            return( Result.toString() );
/* all done            */
          case 13:
/* ignore              */
            break;
          default:
/* real data           */
            Result.append( (char) intChar );
/* append to string    */
          }
/* end of line ?       */
        }
      }
    }
```

The `status` method returns status information about the adapter. In this example it returns for the option `MQe_Adapter_NETWORK` the network type (TCPIP), for the option `MQe_Adapter_LOCALHOST` it returns the tcpip local host address.

```
public String status( Object  opt ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_NETWORK ) )
      return( "TCPIP" );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_LOCALHOST ) )
      return( InetAddress.getLocalHost( ).toString() );
    else
    return( super.status( opt ) );
    }
```

The write method writes a block of data to the socket. It needs to ensure that only one write at a time can be issued to the socket. In this example it calls an internal routine writeBytes to write the actual data and perform any appropriate error recovery.

The opt parameter may be set to:

MQe_Adapter_FLUSH
: flush any data in the buffers

MQe_Adapter_HEADER
: write any header records

MQe_Adapter_HEADERRSP
: write any header response records

```
  public void write( Object opt, int recordSize, byte data[] )
    throws Exception
    {
    synchronized ( writeLock )
  /* only one at a time  */
      {
      if ( checkOption( opt, MQe.MQe_Adapter_HEADER    ) ||
           checkOption( opt, MQe.MQe_Adapter_HEADERRSP )  )
        writeBytes( intToByte( recordSize ), 0, 4 );
  /* write length*/
      writeBytes( data, 0, recordSize );
  /* write the data      */
      if ( checkOption( opt, MQe.MQe_Adapter_FLUSH  ) )
        stream_out.flush( );
  /* make sure it is sent */
      }

    }
```

The writeBytes is an internal method that writes an array (or partial array) of bytes to a socket, and attempt a simple error recovery if errors occur.

```
protected void writeBytes( byte buffer[], int offset, int recordSize )
    throws Exception
    {
    if ( buffer != null )
  /* any data ?          */
      {
      /* break the data up into manageable chuncks */
      int i = 0;
  /* Data index          */
      int j = recordSize;
  /* Data length         */
      int MaxSize    = 4096;
  /* small buffer        */
      int RetryValue = 3;
  /* error retry count   */
      do{
  /* as long as data     */
        if ( j < MaxSize )
  /* smallbuffer ?       */
          MaxSize = j;
        int Retry = RetryValue + 1;
  /* error retry count   */
        do{
  /* possible retry      */
           try
  /* catch io errors     */
            {
            stream_out.write( buffer, offset + i, MaxSize );
            Retry = 0;
```

```
/* don't retry         */
          }
        catch ( IOException e )
/* IO error occured    */
          {
          Retry = Retry - 1;
/* decrement           */
          if ( Retry == 0 )  throw e;
/* more attempts ?     */
          }
        } while ( Retry != 0 );
/* more attempts ?     */

      i = i + MaxSize;
/* update index        */
      j = j - MaxSize;
/* data left           */
      } while ( j > 0 );
/* till all data sent  */
    }
  }
```

The `writeLn` method writes a string of characters to the socket, terminating with 0x0A and 0x0D characters.

The opt parameter may be set to:

MQe_Adapter_FLUSH
flush any data in the buffers

MQe_Adapter_HEADER
write any header records

MQe_Adapter_HEADERRSP
write any header response records

```
public void writeln( Object opt, String data ) throws Exception
  {
  if ( data == null )
/* any data ?          */
    data = "";
  write( opt, -1, MQe.asciiToByte( data + "\r\n" ) );
/* write data  */
  }
```

This is now a complete (though very simple) TCPIP adapter that will communicate to another copy of itself, one of which was started as a listener and the other started as a connector.

An example message store adapter

This example creates an adapter for use as an interface to a message store. It uses the standard Java i/o classes to manipulate files in the store.

This example is not meant as a replacement for the adapters that are supplied with MQe, but rather as a simple introduction to creating a message store adapter.

A new class file is constructed, inheriting from MQeAdapter. Some variables are defined to hold this adapter's instance information, such as the name of the file/message and the location of the message store.

The MQeAdapter constructor is used for the object, so no additional code needs to be added for the constructor.

```
public class MyMsgStoreAdapter extends    MQeAdapter
                               implements FilenameFilter
  {
  protected String  filter   = "";
  /* file type filter    */
  protected String  fileName = "";
  /* disk file name      */
  protected String  filePath = "";
  /* drive and directory */
  protected boolean reading  = false;
/* opened for reading  */
  protected boolean writing  = false;
```

Because this adapter implements FilenameFilter, the following method must be coded. This is the filtering mechanism that is used to select files of a certain type within the message store.

```
  public boolean accept( File dir, String name )
    {
    return( name.endsWith( filter ));
    }
```

Next the `activate` method is coded. This is the method that extracts, from the file descriptor, the name of the directory to be used to hold all the messages.

The Object parameter on the method call may be an attribute object. If it is, this is the attribute that is used to encode and/or decode the messages in the message store.

The Object options for this adapter are:

- MQe_Adapter_READ
- MQe_Adapter_WRITE
- MQe_Adapter_UPDATE

Any other options should be ignored.

```
public void   activate( String  fileDesc,
                        Object  param,
                        Object  options,
                        int     value1,
                        int     value2 ) throws Exception
    {
    super.activate( fileDesc, param, options, lrecl, noRec );
    filePath    = fileId.substring( fileId.indexOf( ':' ) + 1 );
    String Temp = filePath;
  /* copy the path data  */
    if ( filePath.endsWith( File.separator ) )
  /* ending separator ?  */
      Temp      = Temp.substring( 0, Temp.length( ) -
                               File.separator.length( ) );
    else
      filePath  = filePath + File.separator;
  /* add separator       */
    File diskFile = new File( Temp );
    if ( ! diskFile.isDirectory( ) )
  /* directory ?         */
      if ( ! diskFile.mkdirs( ) )
  /* does mkDirs work ?  */
        throw new MQeException( MQe.Except_NotAllowed,
                        "mkdirs '" + filePath + "' failed" );
    filePath = diskFile.getAbsolutePath( ) + File.separator;
    this.open( null );
    }
```

The close method disallows reading or writing.

```
public void close( Object opt ) throws Exception
    {
    reading  = false;
/* not open for reading*/
    writing  = false;
/* not open for writing*/
    }
```

The control method needs to be coded to handle an MQe_Adapter_LIST that is, a request to list all the files in the directory that satisfy the filter. Also to handle an MQe_Adapter_FILTER that is a request to set a filter to control how the files are listed.

```
public Object control( Object opt, Object ctrlObj ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_LIST   ) )
      return( new File( filePath ).list( this ) );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER ) )
      {
      filter = (String) ctrlObj;
  /* set the filter      */
      return( null );
  /* nothing to return   */
      }
    else
    return( super.control( opt, ctrlObj ) );
  /* try ancestor        */
    }
```

The erase method is used to remove a message from the message store.

```
  public void erase( Object opt ) throws Exception
    {
    if ( opt instanceof String )
  /* select file ?       */
      {
      String FN = (String) opt;
  /* re-type the option  */
      if ( FN.indexOf( File.separator ) > -1 )
  /* directory ?         */
        throw new MQeException( MQe.Except_Syntax,
              "Not allowed" );
      if ( ! new File( filePath + FN ).delete( ) )
        throw new MQeException( MQe.Except_NotAllowed,
              "Erase failed" );
      }
    else
      throw new MQeException( MQe.Except_NotSupported,
              "Not supported" );
    }
```

The open method sets the Boolean values that permit either reading of messages or writing of messages.

```
public void open( Object opt ) throws Exception
    {
    this.close( null );
  /* close any open file */
    fileName = null;
  /* clear the filename  */
    if ( opt instanceof String )
  /* select new file ?   */
      fileName = (String) opt;
  /* retype the name     */
```

```
    reading  = checkOption( opt, MQe.MQe_Adapter_READ   ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    writing  = checkOption( opt, MQe.MQe_Adapter_WRITE  ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    }
```

The readObject method reads a message from the message store and recreates an object of the correct type. It also decrypts and decompresses the data if an attribute is supplied on the activate call. This is a special function in that a request to read a file that satisfies the matching criteria specified in the parameter of the read, returns the first message it encounters that satisfies the match.

```
public Object readObject( Object opt ) throws Exception
    {
    if ( reading )
      {
      if ( opt instanceof MQeFields )
        {
        /* 1. list all files in the directory */
        /* 2. read each file in turn and restore as a Fields object */
        /* 3. try an equality check - if equal then return that object */
        String List[] = new File( filePath ).list( this );
        MQeFields Fields = null;
        for ( int i = 0; i < List.length; i = i + 1 )
          try
            {
            fileName = List[i];
      /* remember the name   */
            open( fileName );
      /* try this file       */
            Fields = (MQeFields) readObject( null );
            if ( Fields.equals( (MQeFields) opt ) )
      /* match ?     */
              return( Fields );
            }
          catch ( Exception e )
      /* error occured       */
            {
            }
      /* ignore error        */
        throw new MQeException( Except_NotFound, "No match" );
        }
      /* read the bytes from disk   */
      File diskFile = new File( filePath + fileName );
      byte data[] = new byte[(int) diskFile.length()];
      FileInputStream InputFile = new FileInputStream( diskFile );
      InputFile.read( data );              /* read the file data  */
      InputFile.close( );                  /* finish with file    */
      /* possible Attribute decode of the data       */
      if ( parameter instanceof MQeAttribute )
    /* Attribute encoding ?*/
        data = ((MQeAttribute) parameter).decodeData( null,
                                                      data,
                                                      0,
                                                      data.length );
      MQeFields FieldsObject = MQeFields.reMake( data, null );
      return( FieldsObject );
      }
    else
      throw new MQeException( MQe.Except_NotSupported,
              "Not supported" );
    }
```

The status method returns status information about the adapter. In this examples it can return the filter type or the file name.

```
public String status( Object  opt  ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER   ) )
      return( filter   );
    if ( checkOption( opt, MQe.MQe_Adapter_FILENAME ) )
      return( fileName );
    return( super.status( opt ) );
    }
```

The `writeObject` method writes a message to the message store. It compresses and encrypts the message object if an attribute is supplied on the `activate` method call.

```
public void writeObject( Object  opt,
                           Object  data ) throws Exception
    {
    if ( writing && (data instanceof MQeFields) )
      {
      byte dump[] = ((MQeFields) data).dump( );
  /* dump object */
      /* possible Attribute encode of the data                          */
      if ( parameter instanceof MQeAttribute )
        dump = ((MQeAttribute) parameter).encodeData( null,
                                                      dump,
                                                      0,
                                                      dump.length );
      /* write out the object bytes                                     */
      File diskFile = new File( filePath + fileName );
      FileOutputStream OutputFile = new FileOutputStream( diskFile );
      OutputFile.write( dump );               /* write the data      */
      OutputFile.getFD().sync( );             /* synchronize disk    */
      OutputFile.close();                     /* finish with file    */
      }
    else
      throw new MQeException( MQe.Except_NotSupported, "Not supported" );
    }
```

This is now a complete (though very simple) message store adapter that reads and writes message objects to a message store.

Variations of this adapter could be coded for example to store messages in a database or in nonvolatile memory.

The WebSphere Everyplace Suite (WES) communications adapter

MQe provides sophisticated security that allows applications to run over HTTP, through the protection of an Internet firewall. The purpose of the WebSphere Everyplace communications adapter is to allow MQe applications to authenticate themselves with the WebSphere Everyplace authentication proxy and thus allow messages to flow through it. The following diagram shows a basic scenario with two applications communicating over the Internet through the WebSphere Everyplace authentication proxy.

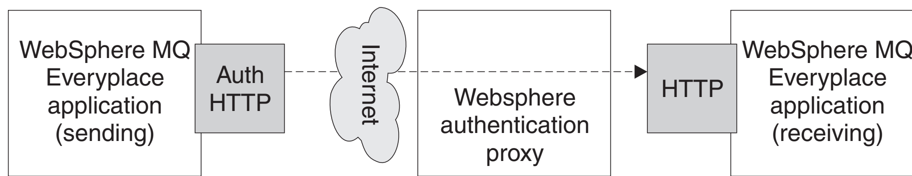


Figure 70. Applications communicating through the WebSphere authentication proxy

The MQe adapter acts as the Auth HTTP adapter on the sending application. The receiving application could use either the same adapter or the standard HTTP adapter provided with MQe.

However, the real value of MQe is that it allows asynchronous messaging to occur in a typically synchronous environment. It is possible to gather enqueued requests from the receiving application and deal with them time-independently. The following diagram shows how incoming requests could be made to reach MQ servers asynchronously.

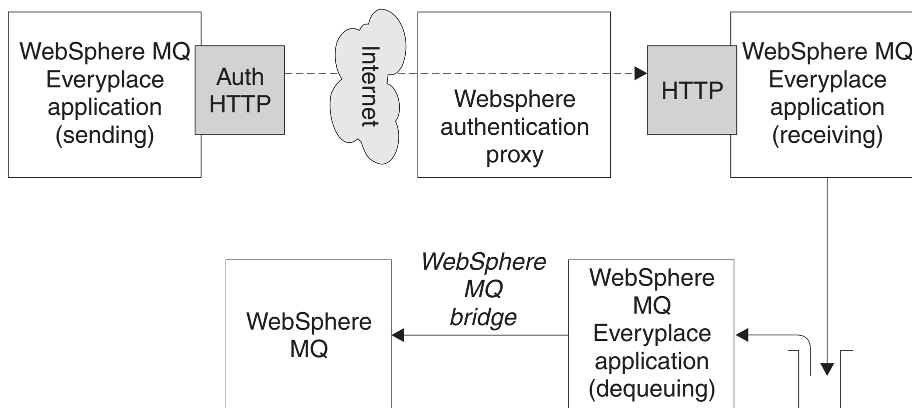


Figure 71. Applications communicating asynchronously through the WebSphere Authentication Proxy

In each of these environments the WebSphere authentication proxy is adding the ability to control access to the receiving applications. The adapter code supports this by adding (application-supplied) user ID and password information to each outgoing HTTP request. The WebSphere authentication proxy accepts these requests and verifies that the supplied credentials are valid for the current environment. If the credentials are valid the proxy forwards the request to the receiving application.

The WebSphere Everyplace Suite (WES) adapter files

In a standard MQe installation the WebSphere Everyplace adapter consists of, and is supported by the following files:

...\Java\com\ibm\mqe\adapters\MQeWESAuthenticationAdapter.class
- The WebSphere Everyplace adapter class.

...\Java\examples\application\Example7.class
- Compiled example application that uses the adapter

...\Java\examples\application\Example7.java
- Source for the example application

...\Java\examples\adapters\WESAuthenticationGUIAdapter.class
- Compiled example adapter that adds a user interface to the WebSphere

Everyplace adapter. As with other example classes, this class is not meant as a replacement for the base WES adapter class, but rather as a demonstration of how to tailor the WES adapter to suit your requirements.

...\Java\examples\adapters\WESAuthenticationGUIAdapter.java
- Source for the example adapter

If your environment *CLASSPATH* variable is set to find all classes within the MQe Java folder, the WebSphere Everyplace adapter class files will be accessible from within the Java environment. If the files are not accessible, issue a command such as:

```
set CLASSPATH=%CLASSPATH%;c:\mqe\java
```

This makes the new classes visible to Java. (The exact format of this command may vary from system to system.) Once this is complete you should be able to use the WebSphere Everyplace adapter classes in the same way as any other MQe classes.

Using the WebSphere Everyplace Suite (WES) adapter

This section provides information on how to use the WebSphere Everyplace adapter. The information is divided into three parts:

General operation
This describes in detail, how to use the adapter in your applications

Using the Authentication Dialog Example
This describes how to use an example class, examples.adapters.WESAuthenticationGUIAdapter. This class is derived from the base WES adapter class and provides a small user interface to collect the ID and password of the user.

Using the Application Example
This describes how to use the supplied example file examples.application.Example7 which is configured to use the base WES adapter.

The information in this section assumes that both the WebSphere Everyplace authentication proxy and MQe have been installed and configured correctly. It is also assumed that an MQe server queue manager and an MQe client queue manager have been configured.

General operation:

1. Configure the client queue manager to send messages using the new adapter by modifying the client queue manager's configuration .ini file so that the *Network* alias points to com.ibm.mqe.adapters.MQeWESAuthenticationAdapter. Use the following command:

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
   ```

2. Configure the server queue manager to decode the stream of data that the Client Adapter supplies using either the new adapter or the standard HTTP adapter. Do this by changing the line in the server queue manager's configuration .ini file so that the *Network* alias points to either com.ibm.mqe.adapters.MQeWESAuthenticationAdapter or com.ibm.mqe.adapters.MQeTcpipHttpAdapter. Use one of the following commands:

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
   ```

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
   ```

3. Modify the client queue manager code so that the required user ID and password are set before the first network operation is started. For example, insert the following line near the top of your code:

```
com.ibm.mqe.adapters.MQeWESAuthenticationAdapter.
setBasicAuthorization("myUserId@myRealm", "myPassword");
```

   Replace the parameters with a valid WES Server user ID and password.

   You also need to add code to catch the new MQeException `Except_Authenticate` after each network operation, in case the supplied credentials were invalid.
4. Check that the client queue manager can still send messages to the server queue manager without going through the proxy.
5. Configure the client machine to send HTTP requests through the proxy. Depending on how WES has been configured, the adapter will need to work with either a *transparent proxy* or an *authentication proxy*.

   As a *transparent proxy*
   In this mode, the WES server acts as a simple HTTP proxy. In this case, you need to set the following Java application system properties that relate to proxy information:

   http.proxyHost
   Must be set to the host name of the WES proxy

   http.proxyPort
   Must be set to the name of the port that the proxy is listening on

   http.proxySet
   Must be set to `true`, which tells the adapter to use transparent proxy mode

   The above parameters can be set by adding the following to your Java application:

```
System.getProperties( ).put( "http.proxySet",  "true" );
System.getProperties( ).put( "http.proxyHost", "wes.hursley.ibm.com" );
System.getProperties( ).put( "http.proxyPort", "8082" );
```

   The client queue manager's connection to the target MQe server is similar to a connection that doesn't use the WES proxy.

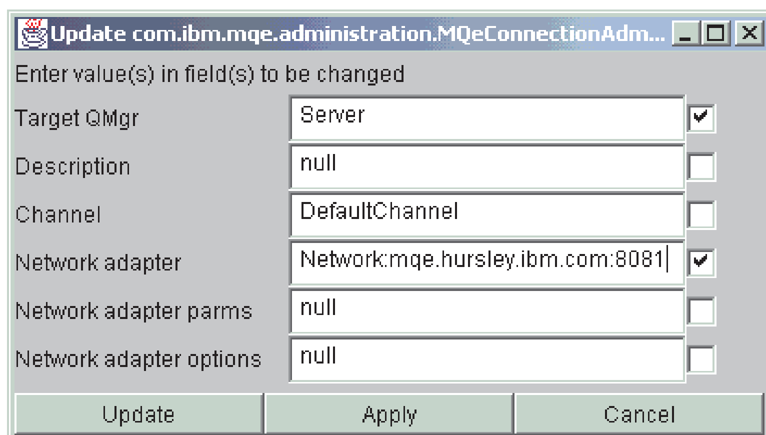


Figure 72. Administration interface panel

You need to restart the server and client queue managers for the new settings to take effect. The client should then be able to send messages to the server through the proxy.

As an *Authentication Proxy*

In this mode, the WES server forwards requests to services, based on the URL that you supply. For example, you may want requests for `http://wes.hursley.ibm.com/mqe` to be forwarded to an MQe queue manager running on `mqe.hursley.ibm.com:8082`.

To set this up from MQe you need to update the client's *`connection`* reference to the server.

Target network adapter
Should point to the Authentication Proxy machine and port

Network adapter parameters
Should contain the pathname to the required service

If you are using the MQe Example Administration tool, select **Connection** and then **Update** to configure this.

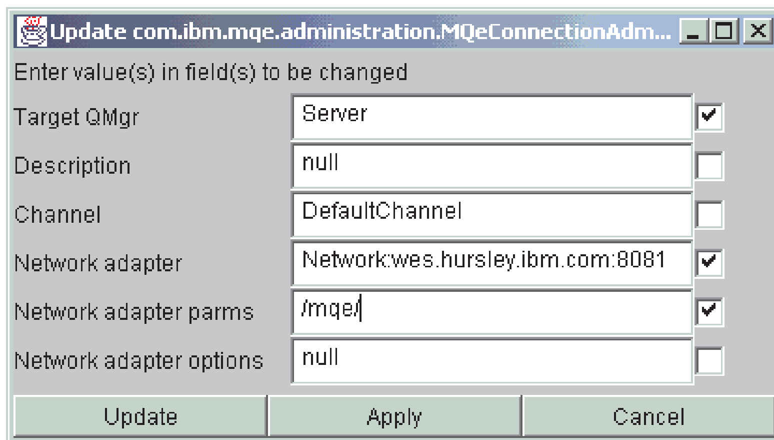


Figure 73. Administration interface panel

Note: The reference to the WES Server is entered in the **Network adapter** field, and the pathname is entered in the **Network adapter parms** field.

You need to restart the server and client queue managers for the new settings to take effect. The client should then be able to send messages to the server through the proxy.

Using the authentication dialog example:

The following information describes the use of the example class file, examples.adapters.WESAuthenticationGUIAdapter. This class adds a small user interface to the base WES adapter function.

1. Follow steps (1) and (2) of the "General operation" on page 154 procedures, but substitute 'WESAuthenticationGUIAdapter' for 'WESAuthenticationAdapter' in step (1).
2. Configure the client's TCP/IP settings as in step (5) of 'General operation'.

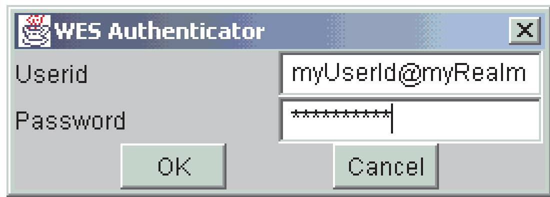


Figure 74. WebSphere Everyplace Suite adapter user dialog

The client should now able to send messages to the server using the WESAuthenticationGUIAdapter. This adapter intercepts write calls to the WES adapter, and on the first request it pops up a dialog box that prompts for user ID and password information.

When the user clicks on **OK** or presses the **Enter** key, the setBasicAuthorization() method is called with the values from the **userid** and **password** fields. The write() is then forwarded on to the underlying WES adapter. The dialog box also has a Cancel button which, when selected, cancels the current write operation by not forwarding the request to the WES adapter. This causes an `MQeException (Except_Stopped)` to be thrown.

If authentication fails, the dialog box is redisplayed on the next write() along with any information provided by the server. In order to learn of an authentication failure, the example adapter intercepts read() calls and catches any `Except_Authenticate` MQeExceptions coming from the adapter.

Note: Web browsers do not generally send authentication information on the first flow. This typically results in a `401` or `407` response that contains the realm information. Only then does the browser send the authenticated request. User clients may wish to follow this convention.

Using the application example:

The following information describes the use of the example application file, examples.application.Example7. This example behaves in a similar way to the MQSeries Everyplace programming example examples.application.Example1 and uses the basic WES adapter for communications.

1. Follow steps (1) and (2) of the "General operation" on page 154 procedures.
2. Configure the client's TCP/IP settings as in step (5) of "General operation" on page 154.
3. Edit the example file ...\Java\examples\application\Example7.java inserting a valid user ID and password, and then recompile the application.
4. Restart the server.
5. Run the Example7 program using the following command:

   ```
   java examples.application.Example7 Server client.ini
   ```

   where

   Server
   : is the name of the remote queue manager (that the client already knows how to reach)

   client.ini
   : points to the client's .ini configuration file.

The application starts the client queue manager, authenticates with the proxy, puts a message to server and then gets a message from the server.

Using rules

Introduction to using MQe rules

MQe uses rules (which are essentially user exits) to allow applications to monitor and modify the behavior of some of its major components. Rules take the form of methods on Java classes or functions in C methods that are loaded when MQe components are initialized.

A component's rules are invoked at certain points during its execution cycle. Rules methods with particular signatures are expected to be available, so when providing implementations of rules, ensure that you use the correct signatures.

Default or example rules are provided for all relevant MQe components. You can customize these to satisfy particular user requirements. Within the Java codebase, the `MQeQueueProxy` interface provides the user with accessor methods for queues, allowing the user to interact with queues in certain rule methods.

Rules may be grouped into the following categories:
- Queue manager rules.
- Queue rules.
- Attribute rules.
- Bridge rules.

Rules may also be categorized into two groups depending upon whether they can affect application behavior (modification rules) or are intended for notification purposes only (notification rules).

Queue manager rules

Queue manager rules are invoked when:
- The queue manager is activated
- The queue manager is closed
- A queue is added to the queue manager (Java codebase only)
- A queue is removed from the queue manager (Java codebase only)
- A put message operation occurs
- A get message operation occurs
- A delete message operation occurs
- An undo message operation occurs
- The queue manager is triggered to transmit any pending messages, as described in Transmission rules

Loading and activating queue manager rules

This topic describes how to load and activate queue manager rules in Java and C.

Java example queue manager rule:

Queue manager rules are loaded, or changed whenever a queue manager administration message containing a request to update the queue manager rule class is received.

If a queue manager rule has already been applied to the queue manager, the existing rule is asked whether it may be replaced with a different rule. If the answer is yes, the new rule is loaded and activated. A restart of the queue manager is not required.

The `QueueManagerUpdater` command-line tool in the package `examples.administration.commandline` shows how to create such an administration message.

C example queue manager rule:

The user's rules module is loaded and initialized when the queue manager is loaded into memory. This occurs as a result of calls either to `mqeAdministrator_QueueManager_create()` or to `mqeQueueManager_new()`. The setup steps are as follows:

- The application must register a rules alias, linking the rules alias to the rules module name and entry point, by using mqeClassAlias_add(), for example:

```
#define RULES_ALIAS "myAlias"
    #define MODULE_NAME "myRulesModule.dll"
    #define ENTRY_POINT "myRules_new"
    ...

    mqeString_newUtf8(pExceptBlock,
                &rulesAlias, RULES_ALIAS);
    mqeString_newUtf8(pExceptBlock,
                &moduleName, MODULE_NAME);
    mqeString_newUtf8(pExceptBlock,
                &entryPoint, ENTRY_POINT);
    mqeClassAlias_add(pExceptBlock,
                rulesAlias, moduleName, entryPoint);
```

- The rules alias must be included in the queue manager start-up parameters passed to either `mqeAdministrator_QueueManager_create()` or `mqeQueueManager_new()`, for example.:

```
MQeQueueManagerParms      qmParams;
    qmParams.hQueueStore = msgStore; /* String parameters for the*/
                          /*location of the msg store */
    qmParams.hQueueManagerRules = rulesAlias; /* add in rules alias */


    /* Indicate what parts of the structure have been set */
    qmParams.opFlags = QMGR_Q_STORE_OP | QMGR_RULES_OP;

    ...

    rc = mqeAdministrator_QueueManager_create(hAdmin,pExceptBlock,
                             &hQM,qmName, &qmParms, &regParms);
```

- An initialization function or entry point must be supplied by the user. The following is an example of an initialization function for a rules implementation. The members of the parameter structures are documented in the MQe C Programming Reference.

```
MQERETURN myRules_new( MQeRulesNew_in_ * pInput,MQeRulesNew_out_ * pOutput) {

    MQERETURN rc = MQERETURN_OK;
    /* declare an instance of the private data */
   /*structure passed around between rules invocations. */
   /*This holds user data which is 'global' between rules. */
```

```
    myRules * myData = NULL;

    /* allocate the memory for the structure */
    myData = malloc(sizeof(myRules));
    if(myData != NULL)    {
   /* map user rules implementations to
      function pointers in output parameter structure */
        pOutput->fPtrActivateQMgr   = myRules_ActivateQMgr;
        pOutput->fPtrCloseQMgr      = myRules_CloseQMgr;
        pOutput->fPtrDeleteMessage  = unitTestRules_DeleteMessage;
        pOutput->fPtrGetMessage     = myRules_getMessage;
        pOutput->fPtrPutMessage     = myRules_putMessage;
        pOutput->fPtrTransmitQueue  = myRules_TransmitQueue;
        pOutput->fPtrTransmitQMgr   = myRules_TransmitQMgr;
        pOutput->fPtrActivateQueue  = myRules_activateQueue;
        pOutput->fPtrCloseQueue     = myRules_CloseQueue;
        pOutput->fPtrMessageExpired = myRules_messageExpired;

        /* initialize data in the private data structure */
        mydata->carryOn = MQE_TRUE;
        mydata->hAdmin = NULL;
        mydata->hThread = NULL;
        mydata->ifp = NULL;
        mydata->triggerInterval = 15000;

        /* now assign the private data structure to */
      /*the output parameter structure variable */
        pOutput->pPrivateData = (MQEVOID *)mydata;
    }
    else {
        /* We had a problem so clear up any strings in the structure -
            none in this case */
    }

    return rc;
}
```

The rules module is unloaded when the queue manager is freed. Note that, unlike the java codebase, the rules implementation is linked to the execution lifecycle of a single queue manager and may not be replaced during the course of this lifecycle.

Using queue manager rules

This topic describes some examples of the use of queue manager rules.

In the Java codebase, a user provides an implementation of a rule method by subclassing the `MQeQueueManagerRule` class.

In the C codebase, a user maps rules functions to relevant rules function pointers. These pointers are passed into the rules initialization function, which is also the entry point to the user's rules module.

For a description of all parameters passed to rules functions in the C codebase, see the MQe C Programming Reference.

Example put message rule:

Put message rule - examples

This first example shows a put message rule that insists that any message being put to a queue using this queue manager must contain an MQe message ID field:

Java codebase

```
/* Only allow msgs containing an ID field to be placed on the Queue */

public void putMessage( String destQMgr, String destQ, MQeMsgObject msg,
                      MQeAttribute attribute, long confirmId )        {
    if ( !(msg.Contains( MQe.Msg_MsgId )) )    {
          throw new MQeException( Except_Rule, "Msg must contain an ID" );
    }
}
```

C codebase

```
MQERETURN myRules_putMessage( MQeRulesPutMessage_in_ * pInput,
                            MQeRulesPutMessage_out_ * pOutput)    {
  // Only allow msgs containing an ID field to be placed on the Queue
  MQERETURN rc = MQERETURN_OK;
  MQEBOOL contains = MQE_FALSE;

  MQeExceptBlock * pExceptBlock=(MQeExceptBlock*)(pOutput->pExceptBlock);
  SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

  rc = mqeFields_contains(pInput->hMsg,pExceptBlock,
                  &contains, MQE_MSG_MSGID);
  if(MQERETURN_OK == rc && !contains)     {
      SET_EXCEPT_BLOCK( pExceptBlock,
                        MQERETURN_RULES_DISALLOWED_BY_RULE,
                        MQEREASON_NA);
  }
}
```

Notice the manner in which the exception block instance is retrieved from the output parameter structure and then set with the appropriate return and reason codes. This is the way in which the rule function communicates with the application, thus modifying application behavior.

Example get message rule:

The next example rule is a get message rule that insists that a password must be supplied before allowing a get message request to be processed on the queue called OutboundQueue. The password is included as a field in the message filter passed into the `getMessage()` method.

Java codebase

```
/* This rule only allows GETs from 'OutboundQueue',
    if a password is  */
/* supplied as part of the filter */

public void getMessage( String destQMgr,
                String destQ, MQeFields filter,
                        MQeAttribute attr, long confirmId )        {
   super.getMessage( destQMgr, destQ, filter, attr, confirmId );
   if (destQMgr.equals(Owner.GetName()
                && destQ.equals("OutboundQueue"))  {
       if ( !(filter.Contains( "Password" ) )      {
           throw new MQeException( Except_Rule,
                        "Password not supplied" );
       }
       else    {
           String pwd = filter.getAscii( "Password" );
           if ( !(pwd.equals( "1234" )) )   {
               throw new MQeException( Except_Rule,
                            "Incorrect password" );
           }
       }
   }
}
```

C codebase

```
MQERETURN myRules_getMessage( MQeRulesGetMessage_in_ * pInput,
                              MQeRulesGetMessage_out_ * pOutput)    {
    MQeStringHndl hQueueManagerName, hCompareString, hCompareString2,
                  hFieldName, hFieldValue;
    MQEBOOL isEqual = MQE_FALSE;
    MQEBOOL contains = MQE_FALSE;
    MQeQueueManagerHndl hQueueManager;

    MQERETURN rc = MQERETURN_OK;
    MQeExceptBlock * pExceptBlock =
                         (MQeExceptBlock *)
                  (pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    /* get the current queue manager */
     rc = mqeQueueManager_getCurrentQueueManager(pExceptBlock,
                                              &hQueueManager);
    if(MQERETURN_OK == rc) {
        // if the destination queue manager is the local queue manager
              rc = mqeQueueManager_getName( hQueueManager,
                                  pExceptBlock,
                                  &hQueueManagerName );
        if(MQERETURN_OK == rc)     {
            rc = mqeString_equalTo(pInput->hQueue_QueueManagerName,
                               pExceptBlock,
                               &isEqual,
                               hQueueManagerName);
            if(MQERETURN_OK == rc && isEqual)    {
                // if the destination queue name is "OutboundQueue"
                rc = mqeString_newUtf8(pExceptBlock,
                                   &hCompareString,
                                   "OutboundQueue");
                rc = mqeString_equalTo(pInput->hQueueName,
                                   pExceptBlock,
                                   &isEqual,
                                   hCompareString);
                if(MQERETURN_OK == rc && isEqual)    {
                    // password required for this queue
                    MQEBOOL contains = MQE_FALSE;
                    rc = mqeString_newUtf8(pExceptBlock,
                                       &hFieldName,
                                       "Password");
                    rc = mqeFields_contains(pInput->hFilter,
                                        pExceptBlock,
                                        &contains,
                                        hFieldName);
                    if(MQERETURN_OK == rc && contains == MQE_FALSE)     {
                        SET_EXCEPT_BLOCK(pExceptBlock,
                                     MQERETURN_RULES_DISALLOWED_BY_RULE,
                                     MQEREASON_NA);
                    }
                    else   {
                            // parse password, etc.
                    }
                }
            }
        }
    }
}
```

This previous rule is a simple example of protecting a queue. However, for more comprehensive security, you are recommended to use an authenticator. An authenticator allows an application to create access control lists, and to determine who is able to get messages from queues.

Example remove queue rule:

The next example rule is called when a queue manager administration request tries to remove a queue. The rule is passed an object reference to the proxy for the queue in question. In this example, the rule checks the name of the queue that is passed, and if the queue is named PayrollQueue, the request to remove the queue is refused.

Java codebase

```
/* This rule prevents the removal of the Payroll Queue */
public void removeQueue( MQeQueueProxy queue )
throws Exception    {
    if ( queue.getQueueName().equals( "PayrollQueue" ) )    {
        throw new MQeException( Except_Rule,
                    "Can't delete this queue" );
    }
}
```

C codebase

This rule is not implemented in the C codebase.

Transmission rules

A message that is put to a remote queue that is defined as synchronous is transmitted immediately. Messages put to remote queues defined as asynchronous are stored within the local queue manager until the queue manager is triggered into transmitting them. The queue manager can be triggered directly by an application. The process can be modified or monitored using the queue manager's transmission rules.

The transmission rules are a subset of the queue manager rules. The two rules that allow control over message transmission are:

`triggerTransmission()`

This rule determines whether to allow message transmission at the time when the rule is called. This can be used to veto or allow the transmission of all messages, that is, either all or none are allowed to be transmitted.

`transmit()`

This rule makes a decision to allow transmission on a per queue basis for asynchronous remote queues. For example, this makes it possible only to transmit the messages from queues deemed to be high priority. The `transmit()` rule is only called if the `triggerTransmission()` rule returns successfully.

Trigger transmission rule example

MQe calls the `triggerTransmission` rule when transmission is triggered. This occurs when the queue manager `triggerTransmission` method or function is explicitly called from an application or a rule. Additionally, in the Java codebase, the rule may be invoked when a message is put onto a remote asynchronous queue. The default rule behavior in both Java and C allows the attempt to transmit pending messages to proceed. For example, this is the default Java rule in com.ibm.mqe.MQeQueueManagerRule:

```
/* default trigger transmission rule -
    always allow transmission */
public boolean triggerTransmission(int noOfMsgs,
      MQeFields msgFields ){
    return true;
}
```

The return code from this rule tells the queue manager whether or not to transmit any pending messages. A return code of true means "transmit", while a return code of false means "do not transmit at this time".

The user may override the default behavior by implementing their own triggerTransmission() rule. A more complex rule can decide whether or not to transmit immediately based on the number of messages awaiting transmission on asynchronous remote queues. The following example shows a rule that only allows transmission to continue if there are more than 10 messages pending transmission.

Java codebase

```
/* Decide to transmit based on number of pending messages */
public boolean triggerTransmission( int noOfMsgs, MQeFields msgFields ) {
    if(noOfMsgs > 10)    {
        return true; /* then transmit */
    }
    else {
        return false; /* else do not transmit */
    }
}
```

C codebase

```
/* The following function is mapped to the
    fPtrTransmitQMgr function pointer  */
/* in the user's initialization function output parameter structure. */

MQERETURN myRules_TransmitQMgr( MQeRulesTransmitQMgr_in_  * pInput,
                        MQeRulesTransmitQMgr_out_ * pOutput)    {
    MQeExceptBlock * pExceptBlock =
                        (MQeExceptBlock*)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

   /* allow transmission to be triggered only
      if the number of pending messages > 10  */
    if(pInput->msgsPendingTransmission <= 10) {
        SET_EXCEPT_BLOCK(pExceptBlock,
                     MQERETURN_RULES_DISALLOWED_BY_RULE,
        MQEREASON_NA);
    }
}
```

Transmit rule

The `transmit()` rule is only called if the `triggerTransmission()` rule allows transmission. It returns a value of true or `MQERETURN_OK`. The `transmit()` rule is called for every remote queue definition that holds messages awaiting transmission. This means that the rule can decide which messages should be transmitted on a queue by queue basis.

A sensible extension to this rule can allow all messages to be transmitted at 'off-peak' time. This allows only messages from high-priority queues to be transmitted during peak periods.

Transmit rule - Java example 1:

The example rule below only allows message transmission from a queue if the queue has a default priority greater than 5. If a message has not been assigned a priority before being placed on a queue, it is given the queue's default priority.

```
public boolean transmit( MQeQueueProxy queue )    {
    if ( queue.getDefaultPriority() > 5 )    {
        return (true);
    }
```

```
    else    {
        return (false);
    }
}
```

Transmit rule - C example 1:

The example rule below only allows message transmission from a queue if the queue has a default priority greater than 5. If a message has not been assigned a priority before being placed on a queue, it is given the queue's default priority.

```
/* The following function is mapped to the fPtrTransmitQueue function*/
/* pointer in the user's initialization
/* function output parameter structure. */

MQERETURN myRules_TransmitQueue( MQeRulesTransmitQueue_in_ * pInput,
                        MQeRulesTransmitQueue_out_ * pOutput) {
    MQERETURN rc = MQERETURN_OK;
    MQEBYTE queuePriority;

    MQeRemoteAsyncQParms queueParms = REMOTE_ASYNC_Q_INIT_VAL;
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
                   (MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    /* inquire upon the default priority of the queue*/
    /* specify the subject of the inquire
      in the queue parameter structure*/
   queueParms.baseParms.opFlags =  QUEUE_PRIORITY_OP ;

    rc =  mqeAdministrator_AsyncRemoteQueue_inquire(myData->hAdmin,
                                    pExceptBlock,
                                    pInput->hQueueName,
                                    pInput->hQueue_QueueManagerName,
                                    &queueParms);
    // if the default priority is less than 6, disallow the operation
    if(MQERETURN_OK == rc
      && queueParms.baseParms.queuePriority < 6) {
        SET_EXCEPT_BLOCK(pExceptBlock,
                        MQERETURN_RULES_DISALLOWED_BY_RULE,
                 MQEREASON_NA);
    }
}
```

A more complex transmit rule example

The following example (in Java and in C) assumes that the transmission of the messages takes place over a communications network that charges for the time taken for transmission. It also assumes that there is a cheap-rate period when the unit-time cost is lower. The rules block any transmission of messages until the cheap-rate period. During the cheap-rate period, the queue manager is triggered at regular intervals.

Transmit rule - Java example 2:

The following example assumes that the transmission of the messages takes place over a communications network that charges for the time taken for transmission. It also assumes that there is a cheap-rate period when the unit-time cost is lower. The rules block any transmission of messages until the cheap-rate period. During the cheap-rate period, the queue manager is triggered at regular intervals.

```
import com.ibm.mqe.*;
import java.util.*;
/**
* Example set of queue manager
    rules which trigger the transmission
* of any messages waiting to be sent.
*
* These rules only trigger the
    transmission of messages if the current
* time is between the values defined
    in the variables cheapRatePeriodStart
* and cheapRatePeriodEnd
* (This example assumes that transmission
      will take place over a
* communication network which charges
      for the time taken to transmit)
*/
public class ExampleQueueManagerRules extends MQeQueueManagerRule
implements Runnable
{
    // default interval between triggers is 15 seconds
    private static final long
          MILLISECS_BETWEEN_TRIGGER_TRANSMITS = 15000;

    // interval between which we c
      heck whether the queue manager is closing down.
        private static final long
          MILLISECS_BETWEEN_CLOSE_CHECKS = 1000 ;

    // Max wait of ten seconds to kil off
        the background thread when
    // the queue manager is closing down.
    private static final long
        MAX_WAIT_FOR_BACKGROUND_THREAD_MILLISECONDS = 10000;

    // Reference to the control block used to
        communicate with the background thread
    // which does a sleep-trigger-sleep-trigger loop.
    // Note that freeing such blocks for garbage
        collection will not stop the thread
    // to which it refers.
    private Thread th = null;

    // Flag which is set when shutdown of
        the background thread is required.
    // Volatile because the thread using the
        flag and the thread setting it to true
    // are different threads, and it is
        important that the flag is not held in
    // CPU registers, or one thread will
        see a different value to the other.
    private volatile boolean toldToStop = false;
   //cheap rate transmission period start and end times
    protected int cheapRatePeriodStart = 18;  /*18:00 hrs */
    protected int cheapRatePeriodEnd = 9;     /*09:00 hrs */
}
```

The cheapRatePeriodStart and cheapRatePeriodEnd functions define the extent of this cheap rate period. In this example, the cheap-rate period is defined as being between 18:00 hours in the evening until 09:00 hours the following morning.

The constant MILLISECS_BETWEEN_TRIGGER_TRANSMITS defines the period of time, in milliseconds, between each triggering of the queue manager. In this example, the trigger interval is defined to be 15 seconds.

The triggering of the queue manager is handled by a background thread that wakes up at the end of the triggerInterval period. If the current time is inside the cheap rate period, it calls the `MQeQueueManager.triggerTransmission()` method to initiate an attempt to transmit all messages awaiting transmission. The background thread is created in the `queueManagerActivate()` rule and stopped in the `queueManagerClose()` rule. The queue manager calls these rules when it is activated and closed respectively.

```
/**
* Overrides MQeQueueManagerRule.queueManagerActivate()
* Starts a timer thread
*/
public void queueManagerActivate()throws Exception {
    super.queueManagerActivate();
    // background thread which triggers transmission
    th = new Thread(this, "TriggerThread");
    toldToStop = false;
    th.start();    // start timer thread
}


/**
* Overrides MQeQueueManagerRule.queueManagerClose()
* Stops the timer thread
*/
 public void queueManagerClose()throws Exception {
    super.queueManagerClose();

    // Tell the background thread to stop,
      as the queue manager is closing now.
    toldToStop = true ;

    // Now wait for the background thread,
      if it's not already stopped.
    if ( th != null)  {
        try {
        // Only wait for a certain time before
        giving up and timing out.
         th.join( MAX_WAIT_FOR_BACKGROUND_THREAD_MILLISECONDS );

         // Free up the thread control block for garbage collection.
            th = null ;
        } catch (InterruptedException e) {
            // Don't propogate the exception.
            // Assume that the thread will stop shortly anyway.
        }
    }
}
```

The code to handle the background thread looks like this:

```
/**
* Timer thread
* Triggers queue manager every interval until thread is stopped
*/
public void run()    {
    /* Do a sleep-trigger-sleep-trigger loop until the */
   /*  queue manager closes or we get an exception.*/
    while ( !toldToStop) {
        try {

            // Count down until we've waited enough
          // We do a tight loop with a smaller granularity because
          // otherwise we would stop a queue manager from closing quickly
          long timeToWait = MILLISECS_BETWEEN_TRIGGER_TRANSMITS ;
          while( timeToWait > 0 && !toldToStop ) {
```

```
            // sleep for specified interval
                Thread.sleep( MILLISECS_BETWEEN_CLOSE_CHECKS );

                // We've waited for some time.
              Account for this in the overall wait.
                timeToWait -= MILLISECS_BETWEEN_CLOSE_CHECKS ;
          }
    if( !toldToStop && timeToTransmit()) {
            // trigger transmission on QMgr (which is rule owner)
            ((MQeQueueManager)owner).triggerTransmission();
                      }
          } catch ( Exception e ) {
              e.printStackTrace();
          }
        }
    }
}
```

The variable owner is defined by the class `MQeRule`, which is the ancestor of `MQeQueueManagerRule`. As part of its startup process, the queue manager activates the queue manager rules and passes a reference to itself to the rules object. This reference is stored in the variable owner.

The thread loops indefinitely, as it is stopped by the `queueManagerClose()` rule, and it sleeps until the end of the `MILLISECS_BETWEEN_TRIGGER_TRANSMITS` interval period. At the end of this interval, if it has not been told to stop, it calls the `timeToTransmit()` method to check if the current time is in the cheap-rate transmission period. If this method succeeds, the queue manager's `triggerTransmission()` rule is called. The `timeToTransmit` method is shown in the following code:

```
protected boolean timeToTransmit()    {
    /* get current time */
    Calendar calendar = Calendar.getInstance();
    calendar.setTime( new Date() );
    /* get hour */
    int hour = calendar.get( Calendar.HOUR_OF_DAY );
    if ( hour >= cheapRatePeriodStart || hour
            < cheapRatePeriodEnd ) {
        return true; /* cheap rate */
    }
    else    {
        return false; /* not cheap rate */
    }
}
```

Transmit rule - C example 2:

The C example emulates the Java example. While the native C codebase is entirely single-threaded, it is possible to write platform-specific code in which threads are created. In this example of a user-written queue manager activate rule, a thread is spawned which loops, sleeping for a period of time defined in a `triggerInterval` variable and then, providing it has not been asked to stop, checking that we are in a cheap rate period prior to attempting to trigger transmission. Data, which is required between rules invocations, is stored in the rule's private data structure. The queue manager's close rule function is used to provide the thread's terminating condition, setting a boolean switch, `carryOn` to `MQE_FALSE`. This switch can be initialized to `MQE_TRUE` in the rules initialization function. This function waits until the thread is suspended before passing control back to the application.

The private data structure passed between rule invocations is as follows:

```
struct myRules_st_ {
// rules instance structure
    MQeAdministratorHndl hAdmin;
// administrator handle to carry around between

// rules functions
    MQEBOOL carryOn;
// used for trigger transmission thread
    MQEINT32 triggerInterval;
// used for trigger transmission thread
    HANDLE hThread;
// handle for the trigger transmission thread
};

typedef struct myRules_st_ myRules;

The queue manager activate rule:

MQEVOID myRules_activateQueueManager( MQeRulesActivateQMgr_in_ * pInput,
                                    MQeRulesActivateQMgr_out_ * pOutput) {
    // retrieve exception block - passed from application
    MQeExceptBlock * pExceptBlock = (MQeExceptBlock *)
                        (pOutput->pExceptBlock);

    // retrieve private data structure passed
      between user's rules invocations
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeQueueManagerHndl hQueueManager;
    MQERETURN rc = MQERETURN_OK;

    rc = mqeQueueManager_getCurrentQueueManager(pExceptBlock,
                              &queueManager);
    if(MQERETURN_OK == rc) {
        // set up the private data administrator
          handle using the retrieved
        // application queue manager handle.
          This is done here rather than in
        // the rules initialization function as the
          queue manager has not yet been
      // activated fully when the rules
      //initialization function is invoked.
        rc = mqeAdministrator_new(pExceptBlock,
                    &myData>hAdmin,hQueueManager);
    }
    if(MQERETURN_OK == rc) {
        DWORD tid;
        // Launch thread to govern calls to trigger transmission
        myData->hThread = (HANDLE) CreateThread(NULL,
                                            0,
                                            timeToTrigger,
                                            (MQEVOID *)myData,
                                            0,
                                            &tId);
                                if(myData>hThread == NULL) {
            // thread creation failed
            SET_EXCEPT_BLOCK(pExceptBlock,
                      MQERETURN_RULES_ERROR,
                      MQEREASON_NA);
        }
    }
}
```

The `timeToTrigger` function provides the equivalent functionality of the `run()` method in the Java example. Notice the use of the private data variable `carryOn`,

type MQEBOOL, as one of the conditions for the while loop to continue. Once this variable has a value of MQE_FALSE, the while loop will terminate, causing the thread to terminate when the function is exited.

```
DWORD _stdcall timeToTrigger(myRules * rulesStruct) {

    MQERETURN rc = MQERETURN_OK;
    MQeQueueManagerHndl hQueueManager;
    MQeExceptBlock exceptBlock;
    myRules * myData = (myRules *)rulesStruct;
    SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);

    /* retrieve the current queue manager */
    rc = mqeQueueManager_getCurrentQueueManager(&exceptBlock,
                                &hQueueManager);
    if(MQERETURN_OK == rc) {
        /* so long as there is not a grave
        internal error and the termination
           condition has not been set */
        while(!(EC(&exceptBlock) ==
                MQERETURN_QUEUE_MANAGER_ERROR &&
                ERC(&exceptBlock) ==
                MQEREASON_INTERNAL_ERROR) &&
                myData->carryOn == MQE_TRUE) {
           /* Are we in a cheap rate transmission period? */
            if(timeToTransmit())    {
                /* if so,  attempt to trigger transmission */
                rc = mqeQueueManager_triggerTransmission(hQueueManager,
                                        &exceptBlock);

                /* wait for the duration of the trigger interval */
                Sleep(myData->triggerInterval);
            }
        }
    }
    return 0;
}
```

The timeToTransmit() function returns a boolean to indicate whether or not we are in a cheap transmission period:

```
MQEBOOL timeToTransmit() {

    SYSTEMTIME timeInfo;
    GetLocalTime(&timeInfo);

    if (timeInfo.wHour >= 18 || timeInfo.wHour < 9) {
        return MQE_TRUE;
    } else {
        return MQE_FALSE;
    }
}
```

It would probably be a better idea to define constants for the cheap rate interval boundary times and carry these around in the rules private data structure also but that has been not been done here for reasons of clarity.

The function returns MQE_TRUE to suggest that we are in a cheap rate period, that is between the hours of 18:00 and 09:00. A return value of MQE_TRUE is one of the prerequisites for transmission to be triggered in timeToTrigger(). Finally, the queue manager close rule is used to terminate the thread. Notice that one of the conditions for termination of the timeToTrigger() function is for the boolean variable carryOn to have a value of MQE_FALSE. In the close function, the value of carryOn is set to false. But, there may still be a considerable lapse of time between when this value is set to MQE_FALSE and when the timeToTrigger() function is

exited. The value of triggerInterval + the time taken to perform a `triggerTransmission` operation. Also, we wait for the thread to terminate in this function. We also call `triggerTransmission()` one more time in case there are still some pending messages.

```
  MQEVOID myRules_CloseQMgr( MQeRulesCloseQMgr_in_ * pInput,
                             MQeRulesCloseQMgr_out_ * pOutput)     {
    MQERETURN rc = MQERETURN_OK;
    MQeQueueManagerHndl hQueueManager;
    myRules * myData = (myRules *)pInput->pPrivateData;
    DWORD result;
    MQeExceptBlock exceptBlock =
              *((MQeExceptBlock *)pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);
    // Effect the ending of the thread by
        setting the MQEBOOL continue to MQE_FALSE
    // This leads to a return from timeToTrigger()
          and hence the implicit call
    // to _endthread
    myData->carryOn = MQE_FALSE;

    /* wait for the thread in any case */
    result = WaitForSingleObject(myData->hThread, INFINITE);

    /* retrieve the current queue manager */
    rc = mqeQueueManager_getCurrentQueueManager(&exceptBlock,
                                  &hQueueManager);
    if(MQERETURN_OK == rc) {
        /* attempt to trigger transmission one
      /* last time to clean up queue */
        rc = mqeQueueManager_triggerTransmission(hQueueManager,
                                  &exceptBlock);
    }
}
```

Activating synchronous remote queue definitions

The queue manager can activate its asynchronous remote queue definitions and home server queues at startup time. In the Java codebase, activating asynchronous remote queue definitions results in an attempt to transmit any messages they contain, while activating home server queues results in an attempt to get any messages that are waiting on their assigned store-and-forward queue. The activateQueues() rule allows this behavior to be configured.

The default rule just returns true.

```
public boolean activateQueues()    {
    return true; /* activate queues on queue manager start-up */
}

/*As with other rules examples above,
    a check can be made to see if the current */
/* time is inside the cheap-rate transmission period.
    This information can then */
/* be used to determine whether queues should be activated or not.

public boolean activateQueues()    {
    if ( timeToTransmit() )    {
        return true;
    }
    else    {
        return false;
    }
}
```

If `activateQueues()` returns false, the remote queue definitions are only activated when a message is put onto them. Home server queues can be activated by calling the queue manager's `triggerTransmission()` method.

In the C codebase, activation of home server queues and asynchronous queues does not result in any attempts to transmit or pull down pending messages. Only explicit calls to the queue manager's `triggerTransmission()` function have this result. There is no implementation of an activateQueues rule in the C codebase. Activation of queues occurs at queue manager startup.

Queue rules

Queue rules

In the Java codebase, each queue has its own set of rules. A solution can extend the behavior of these rules. All queue rules should descend from class com.ibm.mqe.MQeQueueRule.

In the C codebase, only a single set of rules is loaded. A user can implement different rules for different queues by loading other rules modules from the 'master' module. The master rules functions can then invoke the corresponding functions in any other modules as required.

Queue rules are called when:
- The queue is activated.
- The queue is closed.
- A message is placed on the queue using a put operation (Java codebase only).
- A message is removed from the queue using a get operation.
- A message is deleted from the queue using a delete operation (Java codebase only).
- The queue is browsed.
- An undo operation is performed on a message on the queue.
- A message listener is added to the queue (Java codebase only).
- A message listener is removed from the queue (Java codebase only).
- A message expires.
- An attempt is made to change a queue's attributes, that is authenticator, cryptor, compressor (Java codebase only).
- A duplicate message is put onto a queue.
- A message is being transmitted from a remote asynchronouse queue.

Using queue rules

This section describes some examples of the use of queue rules.

The first example shows a possible use of the message expired rule, putting a copy of the message onto a Dead Letter Queue. Both queues and messages can have an expiry interval set. If this interval is exceeded, the message is flagged as being expired. At this point the messageExpired() rule is called. On return from this rule, the expired message is deleted.

Thefirst example sends any expired messages to the queue manager's dead-letter queue, the name of which is defined by the constant `MQe.DeadLetter_Queue_Name` in the Java codebase and`MQE_DEADLETTER_QUEUE_NAME` in the C codebase. The queue

manager rejects a put of a message that has previously been put onto another queue. This protects against a duplicate message being introduced into the MQe network. So, before moving the message to the dead-letter queue, the rule must set the resend flag. This is done by adding the Java `MQe.Msg_Resend` or C `MQE_MSG_RESEND` field to the message.

The message expiry time field must be deleted before moving the message to the dead-letter queue.

Queue rules - Java example 1:

This example shows a possible use of the message expired rule, and a copy of the message is put onto a Dead Letter Queue. Both queues and messages can have an expiry interval set. If this interval is exceeded, the message is flagged as being expired. At this point the messageExpired() rule is called. On return from this rule, the expired message is deleted.

```
/* This rule puts a copy of any expired messages to a Dead Letter Queue */

public boolean messageExpired( MQeFields entry, MQeMsgObject msg )
           throws Exception   {

    /* Get the reference to the Queue Manager */
    MQeQueueManager qmgr = MQeQueueManager.getReference(
                           ((MQeQueueProxy)owner).getQueueManagerName());
    /* need to set re-send flag so that put of message
      to new queue isn't rejected */
    msg.putBoolean( MQe.Msg_Resend, true );
     /* if the message contains an expiry
        interval field - remove it */
    if ( msg.contains( MQe.Msg_ExpireTime )    {
        msg.delete( MQe.Msg_ExpireTime );
    }
    /* put message onto dead letter queue */
    qmgr.putMessage( null, MQe.DeadLetter_Queue_Name,
                msg, null, 0 );
    /* Return true. Note that no use is made
        of this return value - the message is
      always deleted but the return value is kept
      for backward compatibility */
    return (true);
}
```

Queue rules - C example 1:

This example shows a possible use of the message expired rule, and a copy of the message is put onto a Dead Letter Queue. Both queues and messages can have an expiry interval set. If this interval is exceeded, the message is flagged as being expired. At this point the messageExpired() rule is called. On return from this rule, the expired message is deleted.

```
MQEVOID myRules_messageExpired( MQeRulesMessageExpired_in_ * pInput,
                          MQeRulesMessageExpired_out_ * pOutput)  {
    MQERETURN rc = MQERETURN_OK;
    MQeExceptBlock * pExceptBlock =
              (MQeExceptBlock *)(pOutput->pExceptBlock);

    MQEBOOL contains = MQE_FALSE;
    MQeFieldsHndl hMsg;
    MQeQueueManagerHndl hQueueManager;
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    /* Set re-send flag so that attempt to put
      message to new queue isn't rejected */
```

```
    // First, clone the message as the
   //input parameter is read-only
    rc = mqeFields_clone(pInput->hMsg, pExceptBlock,
                    &hMsg);
    if(MQERETURN_OK == rc) {
        rc = mqeFields_putBoolean(hMsg, pExceptBlock,
                                MQE_MSG_RESEND, MQE_TRUE);
        if(MQERETURN_OK == rc) {
            // if the message contains an expiry
            interval field - remove it
            rc = mqeFields_contains(hMsg, pExceptBlock,
                          &contains,
                                    MQE_MSG_EXPIRETIME);
            if(MQERETURN_OK == rc && contains) {
                rc = mqeFields_delete(hMsg, pExceptBlock,
                                      MQE_MSG_EXPIRETIME);
            }
if(MQERETURN_OK == rc) {
                // put message onto dead letter queue
                MQeStringHndl hQueueManagerName;
                rc = mqeQueueManager_getCurrentQueueManager(pExceptBlock,
                                                    &hQueueManager);
                if(MQERETURN_OK == rc) {
            rc = mqeQueueManager_getName(hQueueManager,
                                                pExceptBlock,
                                               &hQueueManagerName);
                    if(MQERETURN_OK == rc) {
                        // use a temporary exception block as don't care
                        // if dead letter queue does not exist
                        MQeExceptBlock tempExceptBlock;
                        SET_EXCEPT_BLOCK_TO_DEFAULT(&tempExceptBlock);
                        rc = mqeQueueManager_putMessage( hQueueManager,
                                                   &tempExceptBlock,
                                                   hQueueManagerName,
                                              MQE_DEADLETTER_QUEUE_NAME,
                                                    hMsg, NULL, 0 );
                       (MQEVOID)mqeString_free(hQueueManagerName,
                                  &tempExceptBlock);
                    }
                }
            }
        }
    }
}
```

Queue rules - Java example 2:

The following example shows how to log an event that occurs on the queue. The event that occurs is the creation of a message listener.

In the example, the queue has its own log file, but it is equally as valid to have a central log file that is used by all queues. The queue needs to open the log file when it is activated, and close the log file when the queue is closed. The queue rules, `queueActivate` and `queueClose` can be used to do this. The variable `logFile` needs to be a class variable so that both rules can access the log file.

```
/* This rule logs the activation of the queue */
public void queueActivate()    {
    try    {
     logFile = new LogToDiskFile( \\log.txt );
     log( MQe_Log_Information, Event_Activate, "Queue " +
     ((MQeQueueProxy)owner).getQueueManagerName() + " + " +
     ((MQeQueueProxy)owner).getQueueName() + " active" );
    }
    catch( Exception e )    {
        e.printStackTrace( System.err );
```

```
    }
}

/* This rule logs the closure of the queue */
public void queueClose()    {
    try    {
      log( MQe_Log_Information, Event_Closed, "Queue " +
          ((MQeQueueProxy)owner).getQueueManagerName() + " + " +
          ((MQeQueueProxy)owner).getQueueName() + " closed" );
        /* close log file */
        logFile.close();
    }
    catch ( Exception e )    {
        e.printStackTrace( System.err );
    }
}
```

The addListener rule is shown in the following code. It uses the MQe.log method to add an Event_Queue_AddMsgListener event.

```
/* This rule logs the addition of a message listener */
public void addListener( MQeMessageListenerInterface listener,
                 MQeFields filter ) throws Exception
   {
    log( MQe_Log_Information, Event_Queue_AddMsgListener,
                "Added listener on queue "
         + ((MQeQueueProxy)owner).getQueueManagerName() + "+"
         + ((MQeQueueProxy)owner).getQueueName() );
}
```

Queue rules - C example 2:

The following example shows how to log an event that occurs on the queue. The event that occurs is a put message request.

In this example, a central log is set up for all queues using the queue activate and close rules. This log is then used to keep track of all putMessage operations. Because the log is shared between rules invocations, the information needed to access the log is stored in the rules private data structure. In this case, the private data structure contains a file handle for passing between rules invocations:

```
struct myRulesData_  {
// rules instance structure
      MQeAdministratorHndl hAdmin;  /
 administrator handle to carry around between
// rules functions
      FILE * ifp;
// file handle for logging rules
};
typedef struct myRulesData_ myRules;
```

In the rules queue activate function, the file is opened and the activation of the queue logged:

```
MQEVOID myRules_activateQueue(MQeRulesActivateQueue_in_ * pInput,
                              MQeRulesActivateQueue_out_ * pOutput)  {
    MQERETURN rc = MQERETURN_OK;
    MQECHAR * qName;
    MQEINT32 size;

    // recover the private data from the input
        structure parameter pInput
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
                (MQeExceptBlock *)(pOutput->pExceptBlock);
```

```
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    if(myData->ifp == NULL)   {
    // initialized to NULL in the rules initialization function
        myData->ifp = fopen("traceFile.txt","w");
        rc =  mqeString_getUtf8(pInput->hQueueName,
                        pExceptBlock, NULL, &size);
        if(MQERETURN_OK == rc) {
             qName = malloc(size);
             rc = mqeString_getUtf8(pInput->hQueueName,
                            pExceptBlock, qName, &size);
             if(MQERETURN_OK ==
              rc && myData->ifp != NULL) {
                 fprintf(myData->ifp,
            "Activating queue %s \n", qName);
             }
        }
    }
}
```

In the rules queue close function, the file is closed after the closure of the queue is logged:

```
MQEVOID myRules_closeQueue(MQeRulesCloseQueue_in_ * pInput,
                           MQeRulesCloseQueue_out_ * pOutput)  {
    MQERETURN rc = MQERETURN_OK;
    MQECHAR * qName;
    MQEINT32 size;

    // recover the private data from the
      input structure parameter pInput
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
          (MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    if(myData->ifp != NULL)   {
        rc =  mqeString_getUtf8(pInput->hQueueName,
                pExceptBlock, NULL, &size);
        if(MQERETURN_OK == rc) {
             qName = malloc(size);
             rc = mqeString_getUtf8(pInput->hQueueName,
                  pExceptBlock, qName, &size);
             if(MQERETURN_OK == rc)  {
                 fprintf(myData->ifp,
              "Closing queue %s \n", qName);
             }
        }
        fclose(myData->ifp);
        MyData->ifp = NULL;
    }
}
```

The rules put message function ensures that each put message operation is logged:

```
MQERETURN myRules_putMessage(MQeRulesPutMessage_in_ * pInput,
                             MQeRulesPutMessage_out_ * pOutput)    {
    MQERETURN rc = MQERETURN_OK;
    MQECHAR * qName, * qMgrName;
    MQEINT32 size;

    // recover the private data from the input structure parameter pInput
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
(MQeExceptBlock *)(pOutput->pExceptBlock);
```

```
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    if(myData->ifp != NULL)   {
        rc =  mqeString_getUtf8(pInput->hQueueName,
                        pExceptBlock, NULL, &size);
        if(MQERETURN_OK == rc) {
             qName = malloc(size);
             rc = mqeString_getUtf8(pInput->hQueueName,
                          pExceptBlock, qName,&size);
        }
        if(MQERETURN_OK == rc)   {
            rc =  mqeString_getUtf8(pInput->hQueue_QueueManagerName,
                          pExceptBlock,
                                      NULL, &size);
            if(MQERETURN_OK == rc) {
                qMgrName = malloc(size);
                rc = mqeString_getUtf8(pInput->hQueue_QueueManagerName,
                                         pExceptBlock,
                            qMgrName, &size);
            }
        }
if(MQERETURN_OK == rc)  {
                 fprintf(myData->ifp, "Putting a message
                  onto queue %s on queue
                         manager %s\n",qName, qMgrName);
        }
    }
    /* allow the operation to proceed regardless of what
      went wrong in this rule */
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);
    return EC(pExceptBlock);
}
```

Java Message Service (JMS)

The MQe classes for Java Message Service (JMS) are a set of Java classes that implement the Sun JMS interfaces to enable JMS programs to access MQe systems. This topic describes how to use the MQe classes for JMS.

The initial release of JMS classes for MQe Version 2.1, supports the point-to-point model of JMS, but does not support the publish or subscribe model.

The use of JMS as the API to write MQe applications has a number of benefits, because JMS is open standard:

- The protection of investment, both in skills and application code
- The availability of people skilled in JMS application programming
- The ability to write messaging applications that are independent of the JMS implementations

More information about the benefits of the JMS API is on Sun's Web site at http://java.sun.com.

Using JMS with MQe

This section describes how to set up your system to run the example programs, including the Installation Verification Test (IVT) example which verifies your MQe JMS installation. To use JMS with MQe you must have the following jar files, in addition to MQeBase.jar, on your class path:

jms.jar
This is Sun’s interface definition for the JMS classes

MQeJMS.jar
This is the MQe implementation of JMS

Obtaining jar files

MQe does not ship with Sun’s JMS interface definition, which is contained in jms.jar, and this must be downloaded before JMS can be used. At the time of writing, this can be freely downloaded fromhttp://java.sun.com/products/jms/docs.htmlThe JMS Version 1.0.2b jar file is required.

In addition, if JMS administered objects are to be stored and retrieved using the Java Naming and Directory Interface (JNDI), the javax.naming.* classes must be on the classpath. If Java 1 is being used, for example, a 1.1.8 JRE, jndi.jar must be obtained and added to the classpath. If Java 2 is being used, a 1.2 or later JRE, the JRE might contain these classes. You can use MQe without JNDI, but at the cost of a small degree of provider dependence.MQe-specific classes must be used for the ConnectionFactory and Destination objects. You can download JNDI jar files from http://java.sun.com/products/jndi

Testing the JMS class path

You can use the example program `examples.jms.MQeJMSIVT` to test your JMS installation. Before you run this program, you need an MQe queue manager that has a `SYSTEM.DEFAULT.LOCAL.QUEUE`. In addition to the JMS jar files mentioned above, you also need the following or equivalent jar files on your class path to run `examples.jms.MQeJMSIVT`:

- MQeBase.jar
- MQeExamples.jar

You can run the example from the command line by typing:

```
java examples.jms.MQeJMSIVT -i
  <ini file name>
```

where <ini file name> is the name of the initialization (ini) file for the MQe queue manager. You can optionally add a ″-t″ flag to turn tracing on:

```
java examples.jms.MQeJMSIVT -t -i
    <ini file name>
```

The example program checks that the required jar files are on the class path by checking for classes that they contain. It creates a *QueueConnectionFactory* and configures it using the ini file name that you passed in on the command line. It starts a connection, which:

1. Starts the MQe queue manager
2. Creates a JMS Queue representing the queue `SYSTEM.DEFAULT.LOCAL.QUEUE` on the queue manager
3. Sends a message to the JMS Queue
4. Reads the message back and compares it to the message it sent

The `SYSTEM.DEFAULT.LOCAL.QUEUE` should not contain any messages before running the program, otherwise the message read back will not be the one that the program sent. The output from the program should look like this:

```
using ini file '<.ini file name>'
    to configure the connection
checking classpath
found JMS interface classes
found MQe JMS classes
found MQe base classes
Creating and configuring QueueConnectionFactory
Creating connection
From the connection data, JMS
  provider is IBM MQe Version 2.0.0.0
Creating session
Creating queue
Creating sender
Creating receiver
Creating message
Sending message
Receiving message


HEADER FIELDS
----------------------------------------
 JMSType:          jms_text
 JMSDeliveryMode:  2
 JMSExpiration:    0
 JMSPriority:      4
 JMSMessageID:     ID:00000009524cf094000000f052fc06ca
 JMSTimestamp:     1032184399562
 JMSCorrelationID: null
 JMSDestination:   null:SYSTEM.DEFAULT.LOCAL.QUEUE
 JMSReplyTo:       null
 JMSRedelivered:   false

 PROPERTY FIELDS (read only)
-----------------------------------------
 JMSXRcvTimestamp : 1032184400133

 MESSAGE BODY (read only)
-----------------------------------------
A simple text message from the MQeJMSIVT program

Retrieved message is a TextMessage; now checking
for equality with the sent message
Messages are equal. Great!
Closing connection
connection closed
IVT finished
```

Running other MQe JMS example programs

MQe provides two other example programs for the JMS classes. The program `examples.jms.PTPSample01` is similar to the IVT examples described above, but there is a command line argument to tell it not to use the *Java Naming and Directory Interface* (JNDI) and it does not have the same checks on the class path. The program requires the same JMS and MQe jar files on the class path as `examples.jms.MQeJMSIVT`, that is jms.jar, MQeJMS.jar, MQeBase.jar, and MQeExamples.jar. It also requires the jndi.jar file, even if it does not use JNDI, because the program imports javax.naming. The section on Using JNDI provides more information on the jndi.jar file. You can run the example from the command line by typing:

```
java examples.jms.PTPSample01 -nojndi -i <ini file name>
```

where <ini file name > is the name of the initialization (ini) file for the MQe queue manager. By default, the program will use the `SYSTEM.DEFAULT.LOCAL.QUEUE` on this queue manager. You can specify a different queue by using the -q flag:

```
java examples.jms.PTPSample01 -i <ini file name> -q <queue name>
```

You can also turn tracing on by adding the -t flag:

```
java examples.jms.PTPSample01 -t -i <ini file name> -q <queue name>
```

The `examples.jms.PTPSample02` program uses message listeners and filters. This program creates a *QueueReceiver* with a ″blue″ filter and creates a message listener for it. It creates a second QueueReceiver with a ″red″ filter and message listener. It then sends four messages to a queue, two with the filter property colour set to blue and two with the filter property colour set to red, and checks that the message listeners receive the correct messages. The program has the same command line parameters as `examples.jms.PTPSample01`.

Writing JMS programs

Introduces the JMS model and provides information on writing MQe JMS applications

This section provides information on writing MQe JMS applications. It provides a brief introduction to the JMS model and information on programming some common tasks that application programs may need to perform.

The JMS model

JMS defines a generic view of a message service. It is important to understand this view, and how it maps onto the underlying MQe system. The generic JMS model is based around the following interfaces that are defined in Sun's javax.jms package:

Connection
This provides a connection to the underlying messaging service and is used to create *Sessions*.

Session
This provides a context for producing and consuming messages, including the methods used to create *MessageProducers* and *MessageConsumers*.

MessageProducer
This is used to send messages.

MessageConsumer
This is used to receive messages.

Destination
This represents a message destination.

Note: A connection is thread safe, but sessions, message producers, and message consumers are not. While the JMS specification allows a Session to be used by more than one thread, it is up to the user to ensure that Session resources are not concurrently used by multiple threads. The recommended strategy is to use one Session per application thread.

Therefore, in MQe terms:

Connection
This provides a connection to an MQe queue manager. All the Connections in a JVM must connect to the same queue manager, because MQe supports a single queue manager per JVM. The first connection created by an application will try and connect to an already running queue manager, and

if that fails will attempt to start a queue manager itself. Subsequent connections will connect to the same queue manager as the first connection.

Session
This does not have an equivalent in MQe

Message producer and message consumer
These do not have direct equivalents in MQe. The MessageProducer invokes the `putMessage()` method on the queue manager. The MessageConsumer invokes the `getMessage()` method on the queue manager.

Destination
This represents an MQe queue.

MQe JMS can put messages to a local queue or an asynchronous remote queue and it can receive messages from a local queue. It cannot put messages to or receive messages from a synchronous remote queue.

The generic JMS interfaces are subclassed into more specific versions for Point-to-point and Publish or Subscribe behavior. MQe implements the Point-to-point subclasses of JMS. The Point-to-point subclasses are:

QueueConnection
Extends Connection

QueueSession
Extends Session

QueueSender
Extends MessageProducer

QueueReceiver
Extends MessageConsumer

Queue
Extends destination

It is recommended that you write application programs that use only references to the interfaces in javax.jms. All vendor-specific information is encapsulated in implementations of:

- QueueConnectionFactory
- Queue

These are known as "administered objects", that is, objects that can be administered and stored in a JNDI namespace. A JMS application can retrieve these objects from the namespace and use them without needing to know which vendor provided the implementation. However, on small devices looking up objects in a JNDI namespace may be impractical or represent an unnecessary overhead. We, therefore, provide two versions of the `QueueConnectionFactory` and Queue classes.

The parent classes, `MQeQueueConnectionFactory.class`, `MQeJMSQueue.class`, provide the base JMS functionality but cannot be stored in JNDI, while subclasses, MQeJNDIQueueConnectionFactory.class, and the MQeJMSJNDIQueue.class, add the necessary functionality for them to be stored and retrieved from JNDI.

Building a connection:

You normally build connections indirectly using a connection factory. A JNDI namespace can store a configured factory, therefore insulating the JMS application from provider-specific information. See the section Using JNDI, below, for details on how to store and retrieve objects using JNDI.

If a JNDI namespace is not available, you can create factory objects at runtime. However, this reduces the portability of the JMS application because it requires references to MQe specific classes. The following code creates a QueueConnectionFactory. The factory uses an MQe queue manager that is configured with an initialisation (ini) file:

```
QueueConnectionFactory factory;
factory = new com.ibm.mqe.jms.MQeJNDIQueueConnectionFactory();
((com.ibm.mqe.jms.MQeJNDIQueueConnectionFactory)factory).
setIniFileName(<initialisation file>)
```

Using the factory to create a connection:

Use the createQueueConnection() to create a QueueConnection:

```
QueueConnection connection;
connection = factory.createQueueConnection();
```

Starting the connection:

Under the JMS specification, connections are not active upon creation. Until the connection starts, MessageConsumers that are associated with the connection cannot receive any messages. Use the following command to start the connection:

```
connection.start();
```

Obtaining a session:

Once a connection has been created, you can use the `createQueueSession()` method on the QueueConnection to obtain a session. The method takes two parameters:

1. A boolean that determines whether the session is ″transacted″ or ″non-transacted″.
2. A parameter that determines the ″acknowledge″ mode. This is used when the session is ″non-transacted″.

The simplest case is that where acknowledgements are used and are handled by JMS itself with `AUTO_ACKNOWLEDGE`, as shown in the following code fragment:

```
QueueSession session;
boolean transacted = false;
session = connection.createQueueSession(transacted, Session.AUTO_ACKNOWLEDGE);
```

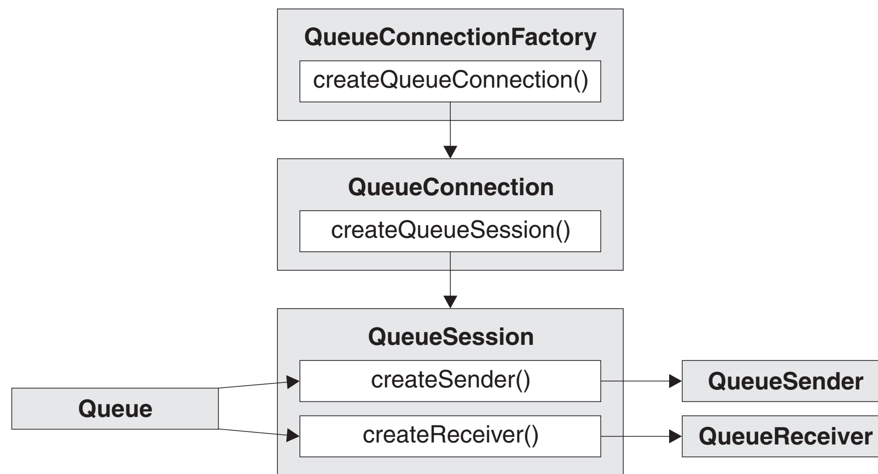


Figure 75. Obtaining a session once a connection is created

Sending a message:

Messages are sent using a MessageProducer. For point-to-point this is a QueueSender that is created using the `createSender()` method on QueueSession. A QueueSender is normally created for a specific Queue, so that all messages sent using that sender are sent to the same destination. Queue objects can be either created at runtime, or built and stored in a JNDI namespace. Refer to "Using Java Naming and Directory Interface (JNDI)" on page 188, for details on how to store and retrieve objects using JNDI.

JMS provides a mechanism to create a Queue at runtime that minimizes the implementation-specific code in the application. This mechanism uses the QueueSession.createQueue() method, which takes a string parameter describing the destination. The string itself is still in an implementation-specific format, but this is a more flexible approach than directly referencing the implementation classes.

For MQe JMS the string is the name of the MQe queue. This can optionally contain the queue manager name. If the queue manager name is included, the queue name is separated from it by a plus sign '+', for example:

```
ioQueue = session.createQueue("myQM+myQueue");
```

This will create a JMS Queue representing the MQe queue "myQueue" on queue manager "myQM". If no queue manager name is specified the local queue manager is used, i.e. the one that JMS is connected to. For example:

```
String queueName = "SYSTEM.DEFAULT.LOCAL.QUEUE";

...

ioQueue = session.createQueue(queueName);
```

This will create a JMS Queue representing the MQe queue `SYSTEM.DEFAULT.LOCAL.QUEUE` on the queue manager that the JMS Connection is using.

Message types:

JMS provides several message types, each of which embodies some knowledge of its content. To avoid referencing the implementation-specific class names for the

message types, methods are provided on the Session object for message creation. In the sample program, a text message is created in the following manner:

```
System.out.println("Creating a TextMessage");
TextMessage outMessage = session.createTextMessage();
System.out.println("Adding Text");
outMessage.setText(outString);
```

The message types that can be used are:
- BytesMessage
- ObjectMessage
- TextMessage

Receiving a message:

Messages are received by using a QueueReceiver. This is created from a Session by using the `createReceiver()` method. This method takes a Queue parameter that defines where the messages are received from. See ″Sending a message″ above for details of how to create a Queue object. The sample program creates a receiver and reads back the test message with the following code:

```
QueueReceiver queueReceiver = session.createReceiver(ioQueue);
Message inMessage = queueReceiver.receive(1000);
```

The parameter in the receive call is a timeout in milliseconds. This parameter defines how long the method should wait if there is no message available immediately. You can omit this parameter, in which case the call blocks indefinitely. If you do not want any delay, use the `receiveNoWait()` method. The receive methods return a message of the appropriate type. For example, if a TextMessage is put on a queue, when the message is received the object that is returned is an instance of TextMessage . To extract the content from the body of the message, it is necessary to cast from the generic Message class, which is the declared return type of the receive methods, to the more specific subclass, such as TextMessage . If the received message type is not known, you can use the ″instanceof″ operator to determine which type it is. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully. The following code illustrates the use of ″instanceof″, and extraction of the content from a TextMessage:

```
if (inMessage instanceof TextMessage){
    String replyString = ((TextMessage)inMessage).getText();
     ...
} else {
    //Print error message if Message was not a TextMessage.
    System.out.println("Reply message was not a TextMessage");
}
```

Handling errors:

Any runtime errors in a JMS application are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and handle them appropriately. Unlike normal Java Exceptions, a JMSException may contain a further exception embedded in it. For JMS, this can be a valuable way to pass important detail from the underlying transport. When a JMSException is thrown as a result of MQe raising an exception, the exception is usually included as the embedded exception in the JMSException. The standard implementation of JMSException does not include the embedded exception in the output of its `toString()` method. Therefore, it is necessary to check explicitly for an embedded exception and print it out, as shown in the following fragment:

```
try {
    ...code which may throw a JMSException
} catch (JMSException je) {
    System.err.println("caught "+je);
    Exception e = je.getLinkedException();
    if (e != null) {
        System.err.println("linked exception:"+e);
    }
}
```

Exception listener:

For asynchronous message delivery, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to `receive()` methods. To cope with this situation, it is possible to register an ExceptionListener, which is an instance of a class that implements the `onException()` method. When a serious error occurs, this method is called with the `JMSException` passed as its only parameter. Further details are in Sun's JMS documentation.

JMS messages:

JMS messages are composed of the following parts:

Header
All messages support the same set of header fields. Header fields contain values that are used by both clients and providers to identify and route messages.

Properties
Each message contains a built-in facility to support application-defined property values. Properties provide an efficient mechanism to filter application-defined messages.

Body JMS defines several types of message body which cover the majority of messaging styles currently in use. JMS defines five types of message body:

Text A message containing a `java.lang.String`

Object
A message that contains a Serializable java object

Bytes A stream of uninterpreted bytes for encoding a body to match an existing message format

Stream
A stream of Java primitive values filled and read sequentially, not supported in this version of MQe JMS

Map A set of name-value pairs, where names are Strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined. Map is not supported in this version of MQe JMS.

The `JMSCorrelationID` header field is used to link one message with another. It typically links a reply message with its requesting message.

Message selectors:

A message contains a built-in facility to support application-defined property values. In effect, this provides a mechanism to add application-specific header fields to a message. Properties allow an application, via message selectors, to have

a JMS provider select or filter messages on its behalf, using application-specific criteria. Application-defined properties must obey the following rules:

- Property names must obey the rules for a message selector identifier.
- Property values can be boolean, byte, short, int, long, float, double, and String.
- The JMSX and JMS_ name prefixes are reserved.

Property values are set before sending a message. When a client receives a message, the message properties are read-only. If a client attempts to set properties at this point, a MessageNotWriteableException is thrown. If `clearProperties()` is called, the properties can then be both read from, and written to.

A property value may duplicate a value in a message's body, or it may not. JMS does not define a policy for what should or should not be made into a property. However, for best performance, applications should only use message properties when they need to customize a message's header. The primary reason for doing this is to support customized message selection. A JMS message selector allows a client to specify the messages that it is interested in by using the message header. Only messages whose headers match the selector are delivered. Message selectors cannot reference message body values. A message selector matches a message when the selector evaluates to true when the message's header field and property values are substituted for their corresponding identifiers in the selector.

A message selector is a String, which can contain:

Literals

- A string literal is enclosed in single quotes. A doubled single quote represents a single quote. Examples are 'literal' and 'literal''s'. Like Java string literals, these use the Unicode character encoding.
- An exact numeric literal is a numeric value without a decimal point, such as 57, -957, +62. Numbers in the range of Java long are supported.
- An approximate numeric literal is a numeric value in scientific notation, such as 7E3 or -57.9E2, or a numeric value with a decimal, such as 7., -95.7, or +6.2. Numbers in the range of Java double are supported. Note that rounding errors may affect the operation of message selectors including approximate numeric literals.
- The boolean literals `TRUE` and `FALSE`.

Identifiers

- An identifier is an unlimited length sequence of Java letters and Java digits, the first of which must be a Java letter. A letter is any character for which the method `Character.isJavaLetter` returns `true`. This includes "_" and "$". A letter or digit is any character for which the method `Character.isJavaLetterOrDigit` returns `true`.
- Identifiers cannot be the names `NULL`, `TRUE`, or `FALSE`.
- Identifiers cannot be `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, and `IS`.
- Identifiers are either header field references or property references.
- Identifiers are case-sensitive.
- Message header field references are restricted to:
  - JMSDeliveryMode
  - JMSPriority
  - JMSMessageID
  - JMSTimestamp

– JMSCorrelationID
– JMSType

JMSMessageID, JMSTimestamp, JMSCorrelationID, and JMSType values may be null, and if so, are treated as a NULL value.

- Any name beginning with ″JMSX″ is a JMS-defined property name
- Any name beginning with ″JMS_″ is a provider-specific property name
- Any name that does not begin with ″JMS″ is an application-specific property name
- If there is a reference to a property that does not exist in a message, its value is NULL. If it does exist, its value is the corresponding property value.

White space
This is the same as is defined for Java, space, horizontal tab, form feed, and line terminator.

Logical operators
Currently supports `AND` only.

Comparison operators
- Only equals ('=') is currently supported.
- Only values of the same type can be compared.
- If there is an attempt to compare different types, the selector is always false.
- Two strings are equal if they contain the same sequence of characters.
- The `IS NULL` comparison operator tests for a null header field value, or a missing property value. The `IS NOT NULL`comparison operator is not supported.

Note that Arithmetic operators are not currently supported.

The following message selector selects messages with a message type of car and a colour of blue: `"JMSType ='car 'AND colour ='blue'"`

When selecting Header fields MQe will interpret exact numeric literals so that they match the type of the field in question, that is a selector testing the JMSPriority or JMSDeliveryMode Header fields will interpret an exact numeric literal as an int, whereas a selector testing JMSExpiration or JMSTimestamp will interpret an exact numeric literal as a long. However, when selecting message properties MQe will always interpret an exact numeric literal as a long and an approximate numeric literal as a double. Application specific properties intended to be used for message selection should therefore be set using the `setLongProperty` and `setDoubleProperty` methods respectively.

Restrictions in this version of MQe

This version of MQe JMS implements the Point-to-Point subset of JMS with a few restrictions. It does not implement any of the optional classes:

- The application server classes ConnectionConsumer, ServerSession, and ServerSessionPool
- The XA classes:
  – XAConnection
  – XAConnectionFactory

- XAQueueConnection
- XAQueueConnectionFactory
- XAQueueSession
- XASession
- XATopicConnection
- XATopicConnectionFactory
- XATopicSession

It does not implement the TemporaryQueue class, which means that the QueueRequestor class will not work or the MapMessage and StreamMessage classes.

In the `QueueConnectionFactory`, the `createQueueConnection()` method that takes a username and password as parameters is not implemented, MQe does not have the concept of a user. The method with no parameters is implemented.

When a message is read from a queue but not acknowledged, the message is returned to the queue for redelivery. In this case the JMSRedelivered header field should be set in the message. MQe JMS does not set this header field.

MQe JMS can put messages to a local queue or an asynchronous remote queue and it can receive messages from a local queue. It cannot put to or receive messages from a synchronous remote queue.

Using Java Naming and Directory Interface (JNDI)

One of the advantages of using JMS is the ability to write applications which are independent of the JMS implementations, allowing you to plug in a JMS implementation which is appropriate for your environment. However, certain JMS objects must be configured in a way which is specific to the JMS implementation you have chosen. These objects are the connection factories and destinations, queues, and they are often referred to as ″administered objects″. In order to keep the application programs independent of the JMS implementation, these objects must be configured outside of the application programs. They would typically be configured and stored in a JNDI namespace. The application would lookup the objects in the namespace and would be able to use them straight away, because they have already been configured.

There may be situations, such as on a small device, where it would not be desirable to use JNDI. In these cases the objects could be configured directly in the application. The cost of not using JNDI would be a small degree of implementation-dependence in the application.

Storing and retrieving objects with JNDI

Before using JNDI to either store or retrieve objects, an ″initial context″ must be set up, as shown in this fragment taken from the MQeJMSIVT_JNDI example program:

```
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;

...
java.util.Hashtable environment =new java.util.Hashtable();
```

```
environment.put(Context.INITIAL_CONTEXT_FACTORY, icf);
environment.put(Context.PROVIDER_URL, url);
Context ctx = new InitialContext(environment );
```

where:

icf
defines a factory class for the JNDI context. This depends upon the JNDI provider that you are using. The documentation supplied by the JNDI provider should tell you what value to use for this. See also the examples below.

url
defines the location of the namespace. This will depend on the type of namespace you are using. If you are using the file system, this will be a file url that identifies a directory in your file system. If you are using LDAP this will be a ldap url that identifies a LDAP server and location in the directory tree of that server. The documentation supplied by the JNDI provider should describe the correct format for the url.

For more details about JNDI usage, see Sun’s JNDI documentation.

Note: Some combinations of the JNDI packages and LDAP service providers can result in an LDAP error 84. To resolve the problem, insert the following line before the call to InitialContext.

```
environment.put(Context.REFERRAL,"throw");
```

Once an initial context is obtained, objects can be stored in and retrieved from the namespace. To store an object, use the `bind()` method:

```
ctx.bind(entryName, object);
```

where ’entryName’ is the name under which you want the object stored, and ’object’ is the object to be stored, for example to store a factory under the name ″ivtQCF″:

```
ctx.bind("ivtQCF", factory);
```

To store an object in a JNDI namespace, the object must satisfy either the javax.naming.Referenceable interface or the java.io.Serializable interface, depending on the JNDI provider you use. The `MQeJNDIQueueConnectionFactory` and `MQeJMSJNDIQueue`classes implement both of these interfaces. To retrieve an object from the namespace, use the`lookup()` method:

```
object = ctx.lookup(entryName);
```

where `entryName` is the name under which you want the object stored , for example, to retrieve a `QueueConnectionFactory` stored under the name ″ivtQCF″:

```
QueueConnectionFactory factory;
factory = (QueueConnectionFactory)ctx.lookup("ivtQCF");
```

Using the sample programs with JNDI

The example program `examples.jms.MQeJMSIVT_JNDI` can be used to test your installation using JNDI. This is very similar to the `examples.jms.MQeJMSIVT` program, except that it uses JNDI to retrieve the connection factory and the queue that it uses. Before you can run this program you must store these two administered objects in a JNDI namespace:

Table 5. Administered objects for a JNDI namespace

Entry name	Java class	Description

Table 5. Administered objects for a JNDI namespace (continued)

ivtQCF	MQeJNDIQueueConnectionFactory	A QueueConnectionFactory configured to use an MQe queue manager
ivtQ	MQeJMSJNDIQueue	A Queue configured to represent an MQe queue which is local to the queue manager used by the ivtQCF entry

The program examples.jms.CreateJNDIEntry or the MQeJMSAdmin tool, explained in the following section, can be used to create these entries. Larger installations may have a *Lightweight Directory Access Protocol (LDAP)* directory available, but for smaller installations a file system namespace may be more appropriate. When you have decided on a namespace you must obtain the corresponding JNDI class files to support the namespace and add these to your classpath. These will vary depending on your choice of namespace and the version of Java you are using.

You must always have the javax.naming.* classes on your classpath. If you are using Java 1 (for example a 1.1.8 JRE) you must obtain a copy of the jndi.jar file and add it to your classpath. If you are using Java 2 (a 1.2 or later JRE) the JRE may contain these classes itself.

If you want to use an LDAP directory, you must obtain JNDI classes that support LDAP, for example Sun's ldap.jar or IBM's ibmjndi.jar, and add these to your classpath. Some Java 2 JREs may already contain Sun's classes for LDAP. See also the section below about LDAP support for Java classes.

If you want to use a file system directory, you must obtain JNDI classes that support the file system, for example Sun's fscontext.jar (which requires providerutil.jar as well) and add these to your classpath. The CreateJNDIEntry example program requires the MQeJMS.jar file on your classpath, in addition to the JNDI jar files. It takes the following command line arguments:

```
java examples.jms.CreateJNDIEntry -url<providerURL>
    [-icf<initialContextFactory>][-ldap]
     [-qcf<entry name><MQe queue manager ini file>]
    [-q<entry name><MQe queue name>]
```

An alternative argument to use is:

```
java  examples.jms.CreateJNDIEntry -h
```

In the previous two examples:

-url<providerURL>
The URL of the JNDI initial context (obligatory parameter)

-icf<initialContextFactory>
The initialContextFactory for JNDI that defaults to the file system:

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

-ldap This should be specified if you are using an LDAP directory

-qcf<entry name><MQe queue manager ini file>
The name of a JNDI entry to be created for a JMS QueueConnectionFactory and the name of an initialisation (ini) file for an MQe queue manager to be used to configure it

-h Displays a help message

The url, -url, must be specified and either a QueueConnectionFactory (-qcf) or a Queue (-q), or both, must be specified. The context factory, -icf, is optional and defaults to a file system directory. The LDAP flag, -ldap, should be specified if an LDAP directory is being used, this prefixes the entry name with "cn=", which is required by LDAP.

For example, if a queue manager with the initialization file *d:\MQe\exampleQM\exampleQM.ini* exists, and you are using a JNDI directory based in the file system at *d:\MQe\data\jndi*, type (all on one line):

```
java  examples.jms.CreateJNDIEntry -url file://d:/MQe/data/jndi -qcf  ivtQCF
d:\MQe\exampleQM\exampleQM.ini
```

Note that forward slashes are used in the url, even if the file system itself uses back slashes. The url directory must already exist. To add an entry for the queue you would type (all on one line):

```
java  examples.jms.CreateJNDIEntry -url file://
d:/MQe/data/jndi -q ivtQ SYSTEM.DEFAULT.LOCAL.QUEUE
```

You could use another local queue instead of the SYSTEM.DEFAULT.LOCAL.QUEUE.

You could also specify the queue name as `exampleQM+SYSTEM.DEFAULT.LOCAL.QUEUE`, where `exampleQM` is the name of the queue manager. If the name of the queue manager is not specified, the local queue manager is used.

Both entries could be added at the same time by typing:

```
java  examples.jms.CreateJNDIEntry
        -url file://d:/MQe/data/jndi -qcf ivtQCF
d:\MQe\exampleQM\exampleQM.ini -q ivtQ
                SYSTEM.DEFAULT.LOCAL.QUEUE
```

Again, you should type all of this command on one line. A maximum of one connection factory and one queue can be added at a time.

When the JNDI entries have been created, you can run the `example .jms.MQeJMSIVT_JNDI` program. This requires the same jar files on the classpath as the MQeJMSIVT program, that is:

- `jms.jar`, Sun's interface definition for the JMS classes
- `MQeJMS.jar`, the MQe implementation of JMS
- `MQeBase.jar`
- `MQeExamples.jar`

It also requires the JNDI jar files, as used for the `CreateJNDIEntry` example program. The example can be run from the command line by typing:

```
java  examples.jms.MQeJMSIVT_JNDI
     -url<providerURL>
```

where <providerURL> is the specified URL of the JNDI initial context. By default the program uses the file system context for JNDI:

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

If necessary you can specify an alternative context:

```
java  examples.jms.MQeJMSIVT_JNDI -url<providerURL>
    -icf<initialContextFactory>
```

You can optionally add a `-t` flag to turn tracing on:

```
java  examples.jms.MQeJMSIVT_JNDI -url<providerURL>
     -icf<initialContextFactory> -t
```

To use the entries in the file system directory created in the CreateJNDIEntry example above, type:

```
java examples.jms.MQeJMSIVT_JNDI -url file://d:/MQe/data/jndi
```

The example program checks that the required jar files are on the classpath by checking for classes that they contain. It looks up the QueueConnectionFactory and the Queue in the JNDI directory. It starts a connection, which starts the MQe queue manager, sends a message to the Queue, reads the message back and compares it to the message it sent. The queue should not contain any messages before running the program, otherwise the message read back will not be the one that the program sent. The first lines of output from the program should look like this:

```
using context factory
      'com.sun.jndi.fscontext.RefFSContextFactory' for the directory
using directory url 'file://d:/MQe/data/jndi'
checking classpath
found JMS interface classes
found MQe JMS classes
found MQe base classes
found jndi.jar classes
found com.sun.jndi.fscontext.RefFSContextFactory classes
Looking up connection factory in jndi
Looking up queue in jndi
Creating connection
```

The rest of the output should be similar to that from the example without JNDI. You can also run the two other example programs, `examples.jms.PTPSample01` and `example .jms.PTPSample02`, using JNDI. These programs requires the same JMS and MQe jar files on the classpath as the `MQeJMSIVT_JNDI` program, that is:

- `jms.jar`
- `MQeJMS.jar`
- `MQeBase.jar`
- `MQeExamples.jar`

They also require the jndi.jar file and the jar files for the JNDI provider you are using, for example, file system or LDAP. The examples can be run from the command line by typing:

```
java  examples.jms.PTPSsample01 -url<providerURL>
```

As in the previous example, `providerURL` is the URL of the JNDI initial context. By default, the program uses the file system context for JNDI, that is`com.sun.jndi.fscontext.RefFSContextFactory`. If necessary you can specify an alternative context:

```
java examples.jms.PTPSsample01 -url<providerURL>
               -icf<initialContextFactory>
```

You can optionally add a ″-t″ flag to turn tracing on: java examples.jms. PTPSsample01 -url <providerURL><-icf initialContextFactory> -t . To use the entries in the file system directory created in the CreateJNDIEntry example above, you would type:

```
java examples.jms.PTPSample01 -url file://d:/MQe/data/jndi
```

The program `examples.jms.PTPSample02` uses message listeners and filters. It creates a `QueueReceiver` with a filter ″colour='blue'″ and creates a message listener for it. It creates a second `QueueReceiver` with a filter ″colour='red'″ and also creates a message listener. It sends four messages to a queue, two with the property ″colour″ set to ″red″ and two with the property ″colour″ set to ″blue″, and checks that the message listeners receive the correct messages. The program has the same command line parameters as the `PTPSample01` program and can be run in the same way. Simply substitute `PTPSample02` for `PTPSample01`.

Mapping JMS messages to MQe messages

This section describes how the JMS message structure is mapped to an MQe message. It is of interest to programmers who wish to transmit messages between JMS and traditional MQe applications.

As described earlier, the JMS specification defines a structured message format consisting of a header, three types of property and five types of message body, while MQe defines a single free-format message object, `MQeMsgObject`. MQe defines some constant field names that messaging applications require, for example UniqueID, MessageID, and Priority, while applications can put data into an MQe message as `<name, value>` pairs.

To send JMS messages using MQe, we define a constant format for storing the information contained in a JMS message within an `MQeMsgObject`. This adds three top-level fields and four `MQeFields` objects to an `MQeMsgObject`, as shown in the following example.

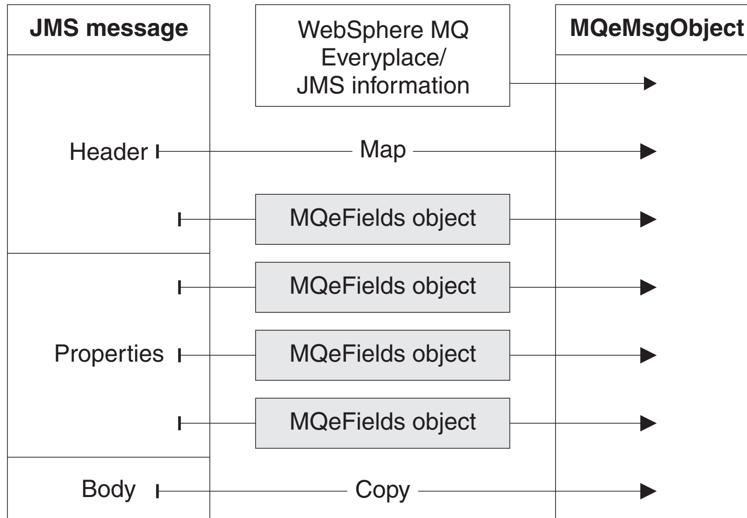


Figure 76. Mapping a JMS message to an MQeMQeMsgObject

The following sections describe the contents of these fields:

Naming MQeMsgObject fields

An `MQeMsgObject` stores data as a `<name, value>` pair. The field names used to map JMS message data to the MQeMsgObject are defined in com.ibm.mqe.MQe and com.ibm.mqe.jms.MQeJMSMsgFieldNames:

MQeJMS field names
: MQe.MQe_JMS_VERSION
  MQeJMSMsgFieldNames.MQe_JMS_CLASS

JMS message field names
: MQeJMSMsgFieldNames.MQe_JMS_HEADER
  MQeJMSMsgFieldNames.MQe_JMS_PROPERTIES
  MQeJMSMsgFieldNames.MQe_JMS_PS_PROPERTIES
  MQeJMSMsgFieldNames.MQe_JMSX_PROPERTIES
  MQeJMSMsgFieldNames.MQe_JMS_BODY

JMS header field names
: MQeJMSMsgFieldNames.MQe_JMS_DESTINATION
  MQeJMSMsgFieldNames.MQe_JMS_DELIVERYMODE
  MQeJMSMsgFieldNames.MQe_JMS_MESSAGEID
  MQeJMSMsgFieldNames.MQe_JMS_TIMESTAMP
  MQeJMSMsgFieldNames.MQe_JMS_CORRELATIONID
  MQeJMSMsgFieldNames.MQe_JMS_REPLYTO
  MQeJMSMsgFieldNames.MQe_JMS_REDELIVERED
  MQeJMSMsgFieldNames.MQe_JMS_TYPE
  MQeJMSMsgFieldNames.MQe_JMS_EXPIRATION
  MQeJMSMsgFieldNames.MQe_JMS_PRIORITY

MQe JMS information

Two <name, value> pairs holding information required for MQe to recreate the JMS message are added directly to the `MQeMsgObject`:

`MQe.MQe_JMS_VERSION`
: This contains a *short* describing the version number of the MQe JMS implementation used to store the message. The current version number is 1. The presence or absence of a field named `MQe.MQe_JMS_VERSION` is used to determine if an `MQeMsgObject` contains an MQe JMS message.

MQeJMSMsgFieldNames.MQe_JMS_CLASS
: This contains a *String* describing the type of JMS message body stored in the `MQeMsgObject`. It defines the strings in the following table:

Table 6. Strings in MQeJMSMsgFieldNames.MQe_JMS_CLASS

JMS message type	MQe.MQe_JMS_CLASS
Bytes message	jms_bytes
Map message	jms_map
Null message	jms_null
Object message	jms_object
Stream message	jms_stream
Text message	jms_text

JMS header files

JMS Header fields are stored within an `MQeMsgObject` using the following rules:

1. If a JMS header field is identical to a defined `MQeMsgObject` field then the header value is mapped directly to the appropriate field in the `MQeMsgObject`.
2. If a JMS header field does not map directly to a defined field but can be represented using existing fields defined by MQe then the JMS header value is converted as appropriate and then set in the `MQeMsgObject`.

3. If MQe has not defined an equivalent field by then, the header field is stored within an MQeFields object, which is then embedded in the MQeMsgObject. This ensures that the JMS header field in question can be restored when the JMS message is recreated.

The header fields that map directly to MQeMsgObject fields are:

Table 7. Header fields that map directly to MQeMsgObject fields

JMS header field	MQeMsgObjectdefined field
JMSTimestamp	MQe.Msg_Time
JMSCorrelationID	MQe.Msg_CorrelID
JMSExpiration	MQe.Msg_ExpireTime
JMSPriority	MQe.Msg_Priority

Two JMS header fields, JMSReplyTo and JMSMessageID, are converted prior to being stored in MQeMsgObject fields.

JMSReplyTo is split between MQe.Msg_ReplyToQMgr and MQe.Msg_ReplyToQ, while JMSMessageID is the String "ID:" followed by a 24-byte hashcode generated from a combination of MQe.Msg_OriginQMgr and MQe.Msg_Time.

The remaining four JMS header fields, JMSDeliveryMode, JMSRedelivered, and JMSType have no equivalents in MQe. These fields are stored within an MQeFields object in the following manner:

- As an int field named MQe.MQe_JMS_DELIVERYMODE
- As a boolean field named MQe.MQe_JMS_REDELIVERED
- As a String field named MQe.MQe_JMS_JMSTYPE

This MQeFields object is then stored within the MQeMsgObject as MQe.MQe_JMS_HEADER. Finally, JMSDestination is recreated when the message is received and, therefore does not need to be stored in the MQeMsgObject.

JMS properties

When storing JMS property fields in an MQeMsgObject, the <name, value> format used by the JMS properties corresponds very closely to the format of data in an MQeFields object:

Table 8. JMS property fields and the MQeFields object

Property type	Corresponding MQeFields object
Application-specific	MQe.MQe_JMS_PROPERTIES
Standard (JMSX_name)	MQe.MQe_JMSX_PROPERTIES
Provider-specific (JMS_provider_name)	MQe.MQe_JMS_PS_PROPERTIES

Three MQeFields objects, corresponding to the three types of JMS property, application-specific, standard, and provider-specific are used to store the <name, value> pairs stored as JMS message properties.

These three MQeFields objects are then embedded in the MQeMsgObject with the following names:

- MQe.MQe_JMS_PROPERTIES, application-specific

- MQe_MQe_JMSX_PROPERTIES, standard properties
- MQe.MQe_JMS_PS_PROPERTIES, provider-specific

Note that MQe does not currently set any provider specific properties. However, this field is used to enable MQe to handle JMS messages from other providers, for example MQ.

The following code fragment creates an MQe JMS text message by adding the required fields to an MQeMsgObject:

```
// create an MQeMsgObject
  MQeMsgObject msg = new MQeMsgObject();

  // set the JMS version number
  msg.putShort(MQe.MQe_JMS_VERSION, (short)1);
  // and set the type of JMS message this MQeMsgObject contains
  msg.putAscii(MQeJMSMsgFieldNames.MQe_JMS_CLASS, "jms_text");

  // set message priority and exipry time - these are mapped to
    JMSPriority and JMSExpiration
  msg.putByte(MQe.Msg_Priority, (byte)7);
  msg.putLong(MQe.Msg_ExpireTime, (long)0);

  // store JMS header fields with no MQe
    equivalents in an MQeFields object
  MQeFields headerFields = new MQeFields();
  headerFields.putBoolean(MQeJMSMsgFieldNames.MQe_JMS_REDELIVERED,
                  false);
  headerFields.putAscii(MQeJMSMsgFieldNames.MQe_JMS_TYPE,
                "testMsg");
  headerFields.putInt(MQeJMSMsgFieldNames.MQe_JMS_DELIVERYMODE,
  Message.DEFAULT_DELIVERY_MODE);
  msg.putFields(MQeJMSMsgFieldNames.MQe_JMS_HEADER,
            headerFields);

  // add an integer application-specific property
  MQeFields propField = new MQeFields();
  propField.putInt("anInt", 12345);
  msg.putFields(MQeJMSMsgFieldNames.MQe_JMS_PROPERTIES,
            propField);

  // the provider-specific and JMSX properties are blank
  msg.putFields(MQeJMSMsgFieldNames.MQe_JMSX_PROPERTIES,
          new MQeFields());
  msg.putFields(MQeJMSMsgFieldNames.MQe_JMS_PS_PROPERTIES,
          new MQeFields());

  // finally add a text message body
  String msgText =
      "A test message to MQe JMS";
  byte[] msgBody = msgText.getBytes("UTF8");
  msg.putArrayOfByte(MQeJMSMsgFieldNames.MQe_JMS_BODY,
                  msgBody);

  // send the message to an MQe Queue
  queueManager.putMessage(null,
                  "SYSTEM.DEFAULT.LOCAL.QUEUE",
                  msg, null, 0);
```

Now, you use JMS to receive the message and print it:

```
// first set up a QueueSession, then...
  Queue queue = session.createQueue
    ("SYSTEM.DEFAULT.LOCAL.QUEUE");
  QueueReceiver receiver = session.createReceiver(queue);
```

```
// receive a message
Message rcvMsg = receiver.receive(1000);

// and print it out
System.out.println(rcvMsg.toString());
```

This gives:

```
  HEADER FIELDS
-----------------------------
  JMSType:          testMsg
  JMSDeliveryMode:  2
  JMSExpiration:    0
  JMSPriority:      7
  JMSMessageID:     ID:00000009524cf094000000f07c3d2266
  JMSTimestamp:     1032876532326
 JMSCorrelationID: null
 JMSDestination:   null:SYSTEM.DEFAULT.LOCAL.QUEUE
  JMSReplyTo:       null
  JMSRedelivered:   false

  PROPERTY FIELDS (read only)
------------------------------
  JMSXRcvTimestamp : 1032876532537
  anInt : 12345

  MESSAGE BODY (read only)
-------------------------------------------------------------------
 A test message to MQe JMS
```

Note that JMS sets some of the JMS message fields, for example `JMSMessageID`, `JMSXRcvTimestamp` internally.

JMS message body:

Regardless of the JMS message type, MQe stores the JMS message body internally as an array of bytes. For the currently supported message types, this byte array is created as follows:

Table 9. JMS message body

JMS message type	Conversion
Bytes message	ByteArrayOutputStream.toByteArray();
Object message	<serialized object>.toByteArray();
Text message	String.getBytes("UTF-8");

When the JMS message body is stored in an `MQeMsgObject`, this *byte* array is added directly to the `MQeMsgObject` with the name `MQe.MQe_JMS_BODY`.

MQe JMS classes

MQe classes for Java Message Service consist of a number of Java classes and interfaces that are based on the Sun javax.jms package of interfaces and classes. They are contained in the com.ibm.mqe.jms package. The following classes are provided:

Table 10. MQe JMS classes

Class	Implements
MQeBytesMessage	BytesMessage

Table 10. MQe JMS classes (continued)

Class	Implements
MQeConnection	Connection
MQeConnectionFactory	ConnectionFactory
MQeConnectionMetaData	ConnectionMetaData
MQeDestination	Destination
MQeJMSEnumeration	Java.util.Enumeration from QueueBrowser
MQeJMSJNDIQueue	Queue
MQeJMSQueue	Queue
MQeMessage	Message
MQeMessageConsumer	MessageConsumer
MQeMessageProducer	MessageProducer
MQeObjectMessage	ObjectMessage
MQeQueueBrowser	QueueBrowser
MQeQueueConnection	QueueConnection
MQeJNDIQueueConnectionFactory	QueueConnectionFactory
MQeQueueConnectionFactory	QueueConnectionFactory
MQeQueueReceiver	QueueReceiver
MQeQueueSender	QueueSender
MQeQueueSession	QueueSession
MQeSession	Session
MQeTextMessage	TextMessage

Note that MessageListener and ExceptionListener are implemented by applications.

Security

Overview of the security features in MQe that enable the protection of data

MQe provides an integrated set of security features that enable the protection of data both when held locally and when it is being transferred.

MQe provides security at several levels:
- Local
- Queue-based
- Message level
- Queue-manager based
- Channel level
- Certificate-based.

MQe also provides the following services to assist with security:
- Private registry services
- Public registry services
- Mini-certificate issuance service.

Levels of security

MQe provides several levels of security:

Local security
Local security provides protection for any MQe data.

Queue-based security
Queue-based security is handled internally by MQe and does not require any specific action by the initiator or recipient of the message.

Message-level security
Message-level security provides protection for message data between an initiating and receiving MQe application.

Queue-manager based security
Security features can be added at the queue-manager level by configuring the queue manager and its private registry.

Channel level security
When data is sent between a queue manager and a remote queue, the queue manager opens a channel to the remote queue manager that owns the queue. By default, if the remote queue is protected, for example with a cryptor, the channel is given exactly the same level of protection as the queue.

Note: Throughout the world there are varying government regulations concerning levels and types of cryptography. You must always use a level and type of cryptography that complies with the appropriate local legislation. This is particularly relevant when using a mobile device that is moved from country to country. MQe provides facilities for this, but it is the responsibility of the application programmer to implement it.

Queue based security is handled internally by MQe and does not require any specific action by the initiator or recipient of the message. Local and Message-level security must be initiated by an application.

All three categories protect Message data by the application of an MQeAttribute, or a descendent. Depending on the category, the attribute is either explicitly or implicitly applied.

Every attribute can contain any or all of the following objects:
- Authenticator
- Cryptor
- Compressor
- Key
- Target Entity Name

The way these objects are used depends on the category of MQe security. Each category of security is described in detail in other topics.

Local security

Local security protects MQe message or MQeFields data locally. This is achieved by creating an attribute with an appropriate symmetric cryptor and compressor, creating and setting up an appropriate *key*, by providing a password. The key is explicitly attached to the attribute, and the attribute is attached to the MQe

message. MQe provides the MQeLocalSecure Java class and C API to assist with the setup of local security, but in all cases it is the responsibility of the local security user (MQe internally or an MQe application) to set up an appropriate attribute and manage the password key.

Local security provides protection for MQe data, MQeFields objects, including Java message objects, for example MQeMsg Object. The protected data is returned in a byte array. To apply local security to a data object you must:

1. Create an attribute with an appropriate authenticator, cryptor, and compressor.
2. Set up an appropriate *key*, by providing a password.
3. Explicitly attach the key to the attribute, the attribute to the data, MQeFields object, and invoke the dump() method on the data object.

The authenticator determines how access to the data is controlled. It is invoked every time a piece of data is acessed. The cryptor determines the cryptographic strength protecting the data confidentiality. The compressor determines the amount of storage required by the message.

MQe provides the MQeLocalSecure class to assist with the use of local security. However, it is the responsibility of the local security user to setup an appropriate attribute and provide the password. MQeLocalSecure provides the function to protect the data and to save and restore it from backing storage. If an application chooses to attach an attribute to a message without using MQeLocalSecure, it also needs to save the data after using dump and must retrieve the data before using restore.

Local security usage scenario:

Consider a scenario where mobile agents working on many different customer sites want to ensure that the confidential data of one customer is not accidentally shared with another. Local security features, using different keys, and possibly different cryptographic strengths, provide a simple method for protecting different customer data held on a single machine .

A simple extension of this scenario could be that the protected local data is accessed using a key that is pulled from a secure queue on an MQe server node. The agent's client has to authenticate itself to access the server queue and pull the local key data, but never knows the actual key.

One of the advantages of taking this approach is that an audit trail is easily accumulated for all access to customer specific data.

Secure feature choices:

When using local security, WebSphere MQ Everyplace provides attribute choices for authentication, encryption, and compression. The algorithms supported by WebSphere MQ Everyplace for authentication, encryption, and compression are listed in the following table:

Table 11. Authentication, encryption and compression support

Function	Algorithm
Authentication	WTLS mini-certificate (NTAuthenticator or UserIdAuthenticator, Java only)
	Validation Windows NT, Windows 2000, AIX®, or Solaris identity
	WinCEAuthenticator (C only)
Compression	LZW (Java only)
	RLE (Java and C)
	GZIP (Java only)
Encryption	Triple DES (Java only)
	DES (Java only)
	MARS (Java only)
	RC4 (Java and C)
	RC6 (Java only)
	XOR (Java only)

You can use your own implementations of authenticators, provided that your cryptor is symmetric.

Selection criteria:

You should use an authenticator if you need to provide additional controls to prevent access to the local data by unauthorized users. In some ways using an authenticator is unnecessary since providing the key password automatically limits access to those who know this secret.

Queue-based security, uses mini-certificate based mutual authentication, and message-level protection.

The choice of cryptor is driven by the strength of protection required. The stronger the encryption, the more difficulty an attacker would face when trying to get illegal access to the data. Data protected with symmetric ciphers that use 128 bit keys is acknowledged as more difficult to attack than data protected using ciphers that use shorter keys. However, in addition to cryptographic strength, the selection of a cryptor may also be driven by many other factors. An example is that some financial solutions require the use of triple DES in order to get audit approval.

You should use a compressor if you need to optimize the size of the protected data. However, the effectiveness of the compressor depends on the content of the data. The Java `MQeRleCompressor` and the C `MQE_RLE_COMPRESSOR` perform run length encoding. This means that the compressor routines compress or expand repeated bytes. Hence it is effective in compressing and decompressing data with many repeated bytes. `MQeLZWCompressor` uses the LZW scheme. The simplest form of the LZW algorithm uses a dictionary data structure in which various words, or data patterns, are stored against different codes. This compressor is likely to be most effective where the data has a significant number of repeating words, or data patterns. The `MQeGZIPCompressor` uses the same compression algorithm as the `gzip`

command on UNIX. This searches for repeating patterns in the data and replaces subsequent occurrences of a pattern with a reference back to the first occurrence of the pattern.

Examples - Java:

1. The following code protects an MQeFields object using MQeLocalSecure

```
 try
{
 .../* SIMPLE UNPROTECT FRAGMENT */
 .../* instantiate a DES cryptor */
 MQeDESCryptor desC = new MQeDESCryptor( );
 .../* instantiate an attribute using the DES cryptor */
 MQeAttribute desA = new MQeAttribute( null, desC, null);
 .../* instantiate a (a helper) LocalSecure object */
 MQeLocalSecure ls = new MQeLocalSecure( );
.../* open LocalSecure obj
    identifying target file and directory */
ls.open( ".\\", "TestSecureData.txt" );
/*instantiate an MQeFields object */
MQeFields myData =new MQeFields();
/*add some test data */
myData.putAscii("testdata","0123456789abcdef....");
.../* use LocalSecure write to protect data*/
ls.write( myData.dump(), desA, "It_is_a_secret" ) );
...
}
catch ( Exception e )
{
e.printStackTrace(); /* show exception */
}

try
    {
    .../* SIMPLE UNPROTECT FRAGMENT     */
    .../* instantiate a DES cryptor     */
    MQeDESCryptor des2C  = new MQeDESCryptor( );
    .../* instantiate an attribute using the DES cryptor  */
    MQeAttribute  des2A  = new MQeAttribute( null, des2C, null);
    .../* instantiate a (a helper) LocalSecure object */
    MQeLocalSecure ls2   = new MQeLocalSecure( );
    .../* open LocalSecure obj identifying
        target file and directory */
    ls2.open( ".\\", "TestSecureData.txt" );
    .../* use LocalSecure read to restore
            from target and decode data*/
    String outData       = MQe.byteToAscii( ls2.read( desA2,
            "It_is_a_secret"));
    .../* show results....  */
    trace ( "i: test data out = " + outData);
    ...
    }
  catch ( Exception e )
    {
    e.printStackTrace();
    /* show exception  */
    }
```

2. The following code protects an MQeMsgObject locally without using MQeLocalSecure.

```
try
  {
   .../*SIMPLE PROTECT FRAGMENT */
   .../*instantiate a DES cryptor */
   MQeDESCryptor desC = new MQeDESCryptor();
   .../*instantiate an Attribute using the DES cryptor */
```

```
        MQeAttribute attr = new MQeAttribute(null,desC,null);
        .../*instantiate a base Key object */
        MQeKey localkey = new MQeKey();
        .../*set the base Key object local key */
        localkey.setLocalKey("my secret key");
        .../*attach the key to the attribute */
        attr.setKey(localkey);
         /*instantiate an MQeFields object */
        MQeFields myData = new MQeFields();
         /*attach the attribute to the data object */
        myData.setAttribute(attr);
         /*add some test data */
        myData.putAscii("testdata", "0123456789abcdef....");
        trace ("i:test data in = " + myData.getAscii("testdata"));
         /*encode the data */
        byte [] protectedData = myData.dump();
        trace ("i:protected test data = " + MQe.byteToAscii(protectedData));
       }
    catch (Exception e )
      {
         e.printStackTrace();  /*show exception */
      }

    try
      {
       .../*SIMPLE UNPROTECT FRAGMENT */
       .../*instantiate a DES cryptor */
      MQeDESCryptor desC2 = new MQeDESCryptor();
       .../*instantiate an Attribute using the DES cryptor */
      MQeAttribute attr2 = new MQeAttribute(null,desC2,null);
       .../*instantiate a base Key object */
      MQeKey localkey2 = new MQeKey();
       .../*set the base Key object local key */
      localkey2.setLocalKey("my secret key");
       .../*attach the key to the attribute */
      attr2.setKey(localkey2 );
       /*instantiate a new data object */
      MQeFields myData2 = new MQeFields();
       /*attach the attribute to the data object */
      myData2.setAttribute(attr2 );
       /*decode the data */
      myData2.restore(protectedData );
       /*show the unprotected test data */
      trace ("i:test data out = " + myData2.getAscii("testdata"));
      }
    catch (Exception e )
      {
       e.printStackTrace();    /*show exception */
       }
```

Examples - C:

1. The following code protects an MQeFields structure using MQeLocalSecure:

```
/* write to a file */
MQeFieldsAttrHndl hAttr = NULL;
MQeStringHndl     hKeySeed = NULL, hDir = NULL, hFile = NULL;
MQeStringHndl     hFieldName = NULL, hFieldData = NULL;
MQeExceptBlock    exceptBlock;
MQeLocalSecureHndl hLocalSecure = NULL;
MQeFieldsHndl     hData = NULL;
MQEBYTE outBuf[128];
MQEINT32 bufLen = 128;

MQERETURN rc;

/* create a key seed string */
rc = mqeString_newChar8(&exceptBlock,
```

```
                 &hKeySeed,
                   "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeFieldsAttr_new(&exceptBlock,
                 hAttr, NULL,
                 MQE_RC4_CRYPTOR_CLASS_NAME,
                 NULL, hKeySeed);
/* create a dir string */
rc = mqeString_newChar8( &exceptBlock, &hDir, ".\\");
/* create a file name string */
rc = mqeString_newChar8( &exceptBlock,
                 &hFile,
                 "localSecureFile.txt");
/* create an MQeLocalSecure */
rc = mqeLocalSecure_new( &exceptBlock, &hLocalSecure);
/* open file */
rc = mqeLocalSecure_open(hLocalSecure, &exceptBlock, hDir, hFile);
/* create a data Fields */
rc = mqeFields_new(&exceptBlock, &hData);
/* add some test data */
rc = mqeString_newChar8(&exceptBlock,
                 &hFieldName,
                 "testdata");
rc = mqeString_newChar8(&exceptBlock,
                 &hFieldData,
                 "0123456789abcdef....");
rc = mqeFields_putAscii(hData, &exceptBlock,
                 hFieldName, hFieldData);
/* dump (protect) data Fields */
rc = mqeFields_dump(hData, &exceptBlock,
               outBuf, &buflen);
/* write to .\\ocalSecureFile.txt */
rc = mqeLocalSecure_write(hLocalSecure, &exceptBlock,
                   outBuf, bufLen, hAttr, NULL);

/* read from a file */
MQeFieldsAttrHndl hAttr = NULL;
MQeStringHndl   hKeySeed = NULL, hDir = NULL, hFile = NULL;
MQeStringHndl   hFieldName = NULL, hFieldData = NULL;
MQeExceptBlock  exceptBlock;
MQeLocalSecureHndl hLocalSecure = NULL;
MQERETURN rc;
MQEBYTE outBuf[128];
MQEINT32 bufLen = 128;

/* create a key seed string */
rc = mqeString_newChar8(&exceptBlock,
           &hKeySeed,
           "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeFieldsAttr_new(&exceptBlock,
           &hAttr, NULL,
           MQE_RC4_CRYPTOR_CLASS_NAME,
           NULL, hKeySeed);
/* create a dir string */
rc = mqeString_newChar8( &exceptBlock,
           &hDir, ".\\");
/* create a file name string */
rc = mqeString_newChar8( &exceptBlock,
           &hFile,
           "localSecureFile.txt");
/* create an MQeLocalSecure */
rc = mqeLocalSecure_new( &exceptBlock,
           &hLocalSecure);
/* open file */
rc = mqeLocalSecure_open(hLocalSecure, &exceptBlock,
           hDir, hFile);
```

```
/* read from .\\ocalSecureFile.txt */
rc = mqeLocalSecure_read(hLocalSecure,
           &exceptBlock, outBuf,
           &Buflen, hAttr, NULL);
/* create a data Fields */
rc = mqeFields_new(&exceptBlock, &hData);
/* restore data Fields */
rc = mqeFields_restore(hData, &exceptBlock,
           outBuf, bufLen);
/* read test data */
rc = mqeString_newChar8(&exceptBlock, &hFieldName,
           "testdata");
rc = mqeFields_getAscii(hData, &exceptBlock,
           &hFieldData, hFieldName);
```

2. The following code protects an MQeFields structure without using MQeLocalSecure:

```
/* dump to a buffer */
MQeFieldsAttrHndl hAttr = NULL;
MQeStringHndl     hKeySeed = NULL, hFieldName =
                   NULL, hFieldData = NULL;
MQeExceptBlock    exceptBlock;
MQeFieldsHndl     hData = NULL;
MQEBYTE           outBuf[128];
MQEINT32          bufLen = 128;
MQERETURN rc;

/* create a key seed string */
rc = mqeString_newChar8(&exceptBlock,
               &hKeySeed,
               "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeFieldsAttr_new(&exceptBlock,
               &hAttr, NULL,
               MQE_RC4_CRYPTOR_CLASS_NAME,
               NULL, hKeySeed);
/* create a data Fields */
rc = mqeFields_new(&exceptBlock, &hData);
/* set the attribute to the data Fields */
rc = mqeFields_setAttribute(hData, &exceptBlock, hAttr);
/* add some test data */
rc = mqeString_newChar8(&exceptBlock,
               &hFieldName,
               "testdata");
rc = mqeString_newChar8(&exceptBlock,
               &hFieldData,
               "0123456789abcdef....");
rc = mqeFields_putAscii(hData, &exceptBlock,
               hFieldName, hFieldData);
/* dump (protect) data Fields */
rc = mqeFields_dump(hData, &exceptBlock,
               outBuf, &bufLen);

/* restor from a buffer */
MQeFieldsAttrHndl hAttr = NULL;
MQeStringHndl     hKeySeed = NULL, hFieldName =
                   NULL, hFieldData = NULL;
MQeExceptBlock    exceptBlock;
MQERETURN rc;
MQEBYTE           outBuf[128];
MQEINT32          bufLen = 128;

...
/* assume protected data is in inBuf
   and its length is in bufLen */

/* create a key seed string */
```

```
rc = mqeString_newChar8(&exceptBlock,
             &hKeySeed,
             "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeFieldsAttr_new(&exceptBlock,
             &hAttr, NULL,
              MQE_RC4_CRYPTOR_CLASS_NAME,
              NULL, hKeySeed);
/* create a data Fields */
rc = mqeFields_new(&exceptBlock, &hData);
/* set the attribute to the data Fields */
rc = mqeFields_setAttribute(hData, &exceptBlock, hAttr);
/* restore data Fields */
rc = mqeFields_restore(hData, &exceptBlock,
             inBuf, bufLen);
/* read test data */
rc = mqeString_newChar8(&exceptBlock,
             &hFieldName, "testdata");
rc = mqeFields_getAscii(hData, &exceptBlock,
             &hFieldData, hFieldName);
```

Message level security

Message-level security facilitates the protection of message data between an initiating and receiving MQe application. Messages are encrypted by the application, using MQe services, and passed to MQe for transport in a fully protected state. MQe delivers the messages to a target queue, from which they are removed by an application and subsequently decrypted, again using MQe services. Since the messages are fully protected when being directly handled by MQe, they can be flowed over clear channels and held on unprotected intermediate queues.

Message-level security is an application layer service. It requires the initiating MQe application to create a message-level attribute and provide it when using **putMessage()** to put a message to a target queue.

The receiving application must set up and pass a matching message-level attribute to the receiving queue manager so that the attribute is available when the application invokes **getMessage()** to get the message from the target queue.

Like local security, message-level security exploits the application of an attribute on a message, an MQeFields object descendent. The initiating application's queue manager handles the application's **putMessage()** with the message Java dump method or C API, which invokes the attached attribute's Java **encodeData()** method or C API to protect the message data. The receiving application's queue manager handles the application's **getMessage()** with the message's Java 'restore' method or C API, which in turn uses the supplied attribute's **decodeData()** method to recover the original message data.

Message-level security usage scenario:

Message-level security is typically most useful for:
- Solutions that are designed to use predominantly asynchronous queues.
- Solutions for which application level security is important; that is solutions whose normal message paths include flows over multiple nodes perhaps connected with different protocols. Message-level security manages trust at the application level, which means security in other layers becomes unnecessary.

A typical scenario is a solution service that is delivered over multiple open networks. For example over a mobile network and the internet, where, from outset

asynchronous operation is anticipated. In this scenario, it is also likely that message data is flowed over multiple links that may have different security features, but whose security features are not necessarily controlled or trusted by the solution owner. In this case it is very likely the solution owner does not want to delegate trust for the confidentiality of message data to any intermediate, but would prefer to manage and control trust management directly.

MQe message-level security provides solution designers with the features that enable the strong protection of message data in a way that is under the direct control of the initiating and recipient applications, and that ensures the confidentiality of the message data throughout its transfer, end to end, application to application.

Secure feature choices:

MQe supplies two alternative attributes for message-level security.

MQeMAttribute
This suits business-to-business communications where mutual trust is tightly managed in the application layer and requires no trusted third party. It allows use of all available MQe symmetric cryptor and compressor choices. Like local security it requires the attribute's key to be preset before it is supplied as a parameter on **putMessage()** and **getMessage()**. This provides a simple and powerful method for message-level protection that enables use of strong encryption to protect message confidentiality, without the overhead of any public key infrastructure (PKI).

MQeMTrustAttribute

Note: The MQeMTrustAttribute does not apply to the C codebase.
This provides a more advanced solution using digital signatures and exploiting the default public key infrastructure to provide a digital envelope style of protection. It uses ISO9796 digital signature or validation so that the receiving application can establish proof that the message came from the purported sender. The supplied attribute's cryptor protects message confidentiality. SHA1 digest guarantees message integrity and RSA encryption and decryption, ensuring that the message can only be restored by the intended recipient. As with MQeMAttribute, it allows use of all available MQe symmetric cryptor and compressor choices. Chosen for size optimization, the certificates used are mini-certificates which conform to the WTLS Specification approved by the WAP forum. MQe provides a default public key infrastructure to distribute the certificates as required to encrypt and authenticate the messages.

A typical MQeMTrustAtribute protected message has the format:

```
RSA-enc{SymKey}, SymKey-enc {Data, DataDigest, DataSignature}
```

where:

RSA-enc:
RSA encrypted with the intended recipient's public key, from his mini-certificate

SymKey:
Generated pseudo-random symmetric key

SymKey-enc:
Symmetrically encrypted with the *SymKey*

Data: Message data

DataDigest:
Digest of message data

DigSignature:
Initiator's digital signature of message data

Selection criteria:

MQeMAttribute relies totally on the solution owner to manage the content of the key seed that is used to derive the symmetric key that is used to protect the confidentiality of the data. This key seed must be provided to both the initiating and recipient applications. While it provides a simple mechanism for the strong protection of message data without the need of any PKI, it clearly depends of the effective operational management of the key seed.

MQeMTrustAttribute exploits the advantages of the MQe default PKI to provide a digital envelope style of message-level protection. This not only protects the confidentiality of the message data flowed, but checks its integrity and enables the initiator to ensure that only the intended recipient can access the data. It also enables the recipient to validate the originator of the data, and ensures that the signer cannot later deny initiating the transaction. This is known as *non-repudiation*.

Solutions that wish to simply protect the end-to-end confidentiality of message data will probably decide that MQeMAttrribute suits their needs, while solutions for which one to one (authenticatable entity to authenticatable entity) transfer and non-repudiation of the message originator are important may find MQeMTrustAttribute is the correct choice.

Examples - using MAttribute for Java:

```
    /*SIMPLE PROTECT FRAGMENT */
{
     MQeMsgObject msgObj = null;
     MQeMAttribute attr = null;
     long confirmId = MQe.uniqueValue();
     try{
        trace(">>>putMessage to target Q using MQeMAttribute"
                +" with 3DES Cryptor and key=my secret key");
         /* create the cryptor */
        MQe3DESCryptor tdes = new MQe3DESCryptor();
         /* create an attribute using the cryptor */
        attr = new MQeMAttribute(null,tdes,null );
         /* create a local key */
        MQeKey localkey = new MQeKey();
         /* give it the key seed */
        localkey.setLocalKey("my secret key");
         /* set the key in the attribute */
        attr.setKey(localkey );
         /* create the message */
        msgObj = new MQeMsgObject();
        msgObj.putAscii("MsgData","0123456789abcdef...");
         /* put the message using the attribute */
        newQM.putMessage(targetQMgrName, targetQName,
            msgObj, attr, confirmId );
        trace(">>>MAttribute protected msg put OK...");
       }
    catch (Exception e)
       {
       trace(">>>on exception try resend exactly once...");
       msgObj.putBoolean(MQe.Msg_Resend, true );
       newQM.putMessage(targetQMgrName, targetQName,
```

```
            msgObj, attr, confirmId );
       }
}

    /*SIMPLE UNPROTECT FRAGMENT */
{
       MQeMsgObject msgObj2 = null;
       MQeMAttribute attr2 = null;
       long confirmId2 = MQe.uniqueValue();
    try{
       trace(">>>getMessage from target Q using MQeMAttribute"+
            " with 3DES Cryptor and key=my secret key");
        /* create the attribute - we do not have to specify the cryptor, */
        /* the attribute can get this from the message itself  */
       attr2 = new MQeMAttribute(null,null,null );
        /* create a local key */
       MQeKey localkey = new MQeKey();
        /* give it the key seed */
       localkey.setLocalKey("my secret key");
        /* set the key in the attribute */
       attr2.setKey(localkey );
        /* get the message using the attribute */
       msgObj2 = newQM.getMessage(targetQMgrName, targetQName,
                                 null, attr2, confirmId2 );
       trace(">>>unprotected MsgData = "
            + msgObj2.getAscii("MsgData"));
       }
    catch (Exception e)
      {
       /*exception may have left */
      newQM.undo(targetQMgrName,
    /*message locked on queue */
                targetQName, confirmId2 );
    /*undo just in case */
      e.printStackTrace();
    /*show exception reason */
      }
     ...
}
```

Examples - using MAttribute for C:

```
    /* putMessage */
   MQeMsgAttrHndl    hAttr = NULL;
   MQeStringHndl     hKeySeed = NULL, hQMgrName =
            NULL, hQName = NULL;
   MQeStringHndl     hFieldName = NULL, hFieldData = NULL;
   MQeExceptBlock    exceptBlock;
   MQeFieldsHndl     hData = NULL;
   MQeQueueManagerHndl hQMgr = NULL;
   MQERETURN rc;

   ...
   /* assume queue manager handle in hQMgr,
  /*QMgr name in hQMgrName, and queue name in hQName */

   /* create a key seed string */
   rc = mqeString_newChar8(&exceptBlock, &hKeySeed,
            "my secret key");
   /* create a new attribute with a RC4 cryptor */
   rc = mqeMsgAttr_new(&exceptBlock, &hAttr, NULL,
            MQE_RC4_CRYPTOR_CLASS_NAME,
            NULL, hKeySeed);
   /* create a data Fields */
   rc = mqeFields_new(&exceptBlock, &hData);
   /* add some test data */
   rc = mqeString_newChar8(&exceptBlock, &hFieldName
```

```
         "MsgData");
rc = mqeString_newChar8(&exceptBlock, &hFieldData
         "0123456789abcdef....");
rc = mqeFields_putAscii(hData, &exceptBlock,
         hFieldName, hFieldData);
/* send message */
rc = mqeQueueManager_putMessage(hQMgr, &exceptBlock,
         hQMgrName, hQName,
         hData, hAttr, 0);

/* getMessage */
MQeMsgAttrHndl hAttr = NULL;
MQeStringHndl     hKeySeed = NULL, hQMgrName =
         NULL, hQName = NULL;
MQeStringHndl     hFieldName = NULL, hFieldData = NULL;
MQeExceptBlock    exceptBlock;
MQeQueueManagerHndl hQMgr = NULL;
MQERETURN rc;

...
/* assume queue manager handle in hQMgr, QMgr
   name in hQMgrName, and queue name in hQName */

/* create a key seed string */
rc = mqeString_newChar8(&exceptBlock, &hKeySeed,
         "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeMsgAttr_new(&exceptBlock, &hAttr, NULL,
         MQE_RC4_CRYPTOR_CLASS_NAME,
         NULL, hKeySeed);
/* get message */
rc = mqeQueueManager_getMessage(hQMgr, &exceptBlock,
         &hData, hQMgrName,
         hQName, NULL, hAttr, 0);
/* get test data */
rc = mqeString_newChar8(&exceptBlock, &hFieldName,
         "MsgData");
rc = mqeFields_getAscii(hData, &exceptBlock,
         &hFieldData, hFieldName);
```

Examples - using MTrustAttribute for Java:

For an explanation about MQePrivateRegistry and MQePublicRegistry (used in the following example) see “Private registry service” on page 236and “Public registry service” on page 240.

```
    /*SIMPLE PROTECT FRAGMENT */
{
    MQeMsgObject msgObj = null;
    MQeMTrustAttribute attr = null;
    long confirmId = MQe.uniqueValue();
try {
    trace(">>>putMessage from Bruce1 intended for Bruce8"
         + " to target Q using MQeMTrustAttribute
          with MARSCryptor ");
     /* create the cryptor */
    MQeMARSCryptor mars = new MQeMARSCryptor();
     /* create an attribute using the cryptor */
    attr = new MQeMTrustAttribute(null, mars, null);
     /* open the private registry belonging to the sender */
    String EntityName = "Bruce1";
    String PIN = "12345678";
    Object Passwd = "It_is_a_secret";
    MQePrivateRegistry sendreg = new MQePrivateRegistry();
    sendreg.activate(EntityName, ".\\MQeNode_PrivateRegistry",
                     PIN, Passwd, null, null );
     /* set the private registry in the attribute */
```

```
    attr.setPrivateRegistry(sendreg );
     /* set the target (recipient) name in the attribute */
    attr.setTarget("Bruce8");
     /* open a public registry to get the target's certificate */
    MQePublicRegistry pr = new MQePublicRegistry();
    pr.activate("MQeNode_PublicRegistry", ".\\");
     /* set the public registry in the attribute */
    attr.setPublicRegistry(pr);
     /* set a home server, which is used to find the certificate*/
     /* if it is not already in the public registry */
    attr.setHomeServer(MyHomeServer +":8082");
     /* create the message */
    msgObj =new MQeMsgObject();
    msgObj.putAscii("MsgData","0123456789abcdef...");
     /* put the message using the attribute */
    newQM.putMessage(targetQMgrName, targetQName,
                     msgObj, attr, confirmId );
    trace(">>>MTrustAttribute protected msg put OK...");
    }
catch (Exception e)
    {
    trace(">>>on exception try resend exactly once...");
    msgObj.putBoolean(MQe.Msg_Resend, true);
    newQM.putMessage(targetQMgrName, targetQName,
                     msgObj, attr, confirmId );
    }
}


    /*SIMPLE UNPROTECT FRAGMENT */
{
    MQeMsgObject msgObj2 = null;
    MQeMTrustAttribute attr2 = null;
    long confirmId2 = MQe.uniqueValue();
try {
    trace(">>>getMessage from Bruce1 intended for Bruce8"
         + " from target Q using MQeMTrustAttribute with MARSCryptor ");
     /* create the cryptor */
    MQeMARSCryptor mars = new MQeMARSCryptor();
     /* create an attribute using the cryptor */
    attr2 = new MQeMTrustAttribute(null, mars, null);
     /* open the private registry belonging to the target */
    String EntityName =  "Bruce8";
    String PIN =  "12345678";
    Object Passwd =  "It_is_a_secret";
    MQePrivateRegistry getreg = new MQePrivateRegistry();
    getreg.activate(EntityName, ".\\MQeNode_PrivateRegistry",
                    PIN, Passwd, null, null );
     /* set the private registry in the attribute */
    attr2.setPrivateRegistry(getreg);
     /* open a public registry to get the sender's certificate */
    MQePublicRegistry pr = new MQePublicRegistry();
    pr.activate("MQeNode_PublicRegistry", ".\\");
     /* set the public registry in the attribute */
    attr2.setPublicRegistry(pr);
     /* set a home server, which is used to find the certificate*/
     /* if it is not already in the public registry */
    attr2.setHomeServer(MyHomeServer +":8082");
     /* get the message using the attribute */
    msgObj2 = newQM.getMessage(targetQMgrName,
                               targetQName, null, attr2, confirmId2 );
    trace(">>>MTrustAttribute protected msg = "
          + msgObj2.getAscii("MsgData"));
    }
catch (Exception e)
    {
     /*exception may have left */
```

```
    newQM.undo(targetQMgrName,             /*message locked on queue */
              targetQName, confirmId2 );  /*undo just in case */
    e.printStackTrace();                 /*show exception reason */
    }
}
```

Non-repudiation:

The MQeMTrustAttribute digitally signs messages. This enables the recipient to validate the creator of the message, and ensures that the creator cannot later deny creating the message. This is known as *non-repudiation*. This process depends on the fact that only one public key can validate the signature successfully generated by a particular private key. This validation proves that the signature was created with the corresponding private key. The only way the alleged creator can deny creating the message is to claim that someone else had access to the private key.

When a message is created with the MQeMTrustAttribute, it uses the private key from the sender's private registry to create the digital signature and it stores the sender's name in the message. When the message is read with the queue manager's **getMessage()** method, it uses the sender's public certificate to validate the digital signature. The message is read successfully only if the signature validates successfully, proving that the message was created by the entity whose name was stored in the message as the sender.

When the MQeMTrustAttribute is specified as a parameter to the queue manager's **getMessage()** method, the attribute validates the digital signature but by the time the message is returned to the user's application all the information relating to the signature has been discarded. If non-repudiation is important to you, you must keep a record of this information. The simplest way to do this is to keep a copy of the encrypted message, because that includes the digital signature. You can do this by using the **getMessage()** method without an attribute. This returns the encrypted message which you can then save, for example in a local queue. You can decrypt the message by applying the attribute to access the contents of the message.

Example - saving a copy of an encrypted message:

The following code fragment provides an example of how to save an encrypted message.

```
/*SIMPLE FRAGMENT TO SAVE ENCRYPTED MESSAGE*/
{
MQeMsgObject msgObj2 = null;
MQeMTrustAttribute attr2 = null;
long confirmId2 = MQe.uniqueValue();
long confirmId3 = MQe.uniqueValue();
try {
    trace(">>>getMessage from Bruce1
        intended for Bruce8"
    + " from target Q using MQeMTrustAttribute
      with MARSCryptor ");
    /* read the encrypted message without an attribute */
    MQeMsgObject tmpMsg1 = newQM.getMessage(targetQMgrName,
        targetQName, null, null, confirmId2 );
    /* save the encrypted message -
      we cannot put it directly */
    /* to another queue because of
      the origin queue manager  */
    /* data. Embed it in another message */
    MQeMsgObject tmpMsg2 = new MQeMsgObject();
    tmpMsg2.putFields("encryptedMsg", tmpMsg1);
    newQM.putMessage(localQMgrName, archiveQName,
```

```
        tmpMsg2, null, confirmId3);
    trace(">>>encrypted message saved locally");
    /* now decrypt and read the message & */
    /* create the cryptor */
    MQeMARSCryptor mars = new MQeMARSCryptor();
    /* create an attribute using the cryptor */
    attr2 = new MQeMTrustAttribute(null, mars, null);
    /* open the private registry belonging to the target */
    String EntityName =  "Bruce8";
    String PIN =  "12345678";
    Object Passwd =  "It_is_a_secret";
    MQePrivateRegistry getreg = new MQePrivateRegistry();
    getreg.activate(EntityName,
      ".\\MQeNode_PrivateRegistry",
      PIN, Passwd, null, null );
    /* set the private registry in the attribute */
    attr2.setPrivateRegistry(getreg);
    /* open a public registry to
      get the sender's certificate */
    MQePublicRegistry pr = new MQePublicRegistry();
    pr.activate("MQeNode_PublicRegistry", ".\\");
    /* set the public registry in the attribute */
    attr2.setPublicRegistry(pr);
    /* set a home server, which is
      used to find the certificate*/
    /* if it is not already in the public registry */
    attr2.setHomeServer(MyHomeServer +":8082");
    /* decrypt the message by unwrapping it */
    msgObj2 = tmpMsg1.unwrapMsgObject(attr2);
    trace(">>>MTrustAttribute protected msg = "
    + msgObj2.getAscii("MsgData"));

catch (Exception e)
  {      /*exception may have left */
    newQM.undo(targetQMgrName,
    /*message locked on queue */
         targetQName, confirmId2 );
    /*undo just in case */
    e.printStackTrace();
    /*show exception reason */
   }
}
```

Queue-based security

Queue-based security is handled internally by MQe and does not require any specific action by the initiator or recipient of the message. Messages are assumed to have been encrypted by the application when they are passed to MQe. MQe delivers the messages to a target queue, from which they are removed by an application.

MQe protects the messages on receipt and flows them over secure channels; they are also held protected on any intermediate queues and on the destination queue. This protection is independent of whether the target queue is owned by a local or a remote queue manager.

Using queue-based security does not require any application programming, but the following topic (“Configuring queue-based security” on page 214) describes how to add security attributes to a queue. As long as configurations have been set up properly, messages are automatically protected during transmission.

Security properties:

The level of queue-based security to be used is determined through the setting of attributes on queues. As a consequence of these attributes, MQe uses, if required, appropriate secure channels, cryptors, and compressors, and controls access through authenticators.

The relevant queue properties are:

Compressor
: A compressor is optional. It determines whether the data should be compressed.

Cryptor
: A cryptor is optional. It determines whether the data should be encrypted to hide the significance of the contents.

Authenticator
: An authenticator is optional. It determines whether the data access should be controlled.

Attribute rule
: An attribute rule is optional in the sense that you can specify a null for this property. If a null is specified, a system default attribute rule is then used internally. An attribute rule determines whether an existing channel can be reused or upgraded to access a particular queue.

Effects of queue attributes:

Queue attributes can be set on all queue definitions. They affect not only the way messages are stored on the queues in question, but also affect the way messages are transmitted over communication channels. MQe creates security attributes internally based on target queue attributes. The effect they have depends upon the kind of queue definition the queue attributes relate to:

Local queue
: Determines how the data is stored and whether the incoming channel characteristics are acceptable. If an authenticator is specified, an authentication process using this authenticator occurs when the queue is accessed for the first time by any particular instance of a local queue manager.

Remote queue
: Determines how the data is stored pending transmission, if applicable, and how the outgoing channel is established. If an authenticator is specified, an authentication process using this authenticator occurs whenever a new channel for transmitting messages on the queue is created.

Store-and-forward queue
: Determines how the data is stored pending transmission, whether the incoming channel characteristics are acceptable, and how the outgoing channel is established (if applicable). An authenticator on a store-and-forward queue has the same effect that it has on a remote queue.

Home server queue
: Determines how the outgoing channel is established. An authenticator on a home-server queue has the same effect that it has on a remote queue.

Configuring queue-based security:

This topic explains how to add security attributes to a queue.

Writing authenticators:

Authenticators are invoked by security attributes. Therefore, how and when they are used is determined by the specific implementation of an attribute. One main usage of authenticators is for controlling access to queues in queue-based security. Authenticators can be used in queue-based security to control access to queues. MQe provides a certificate authenticator as part of its base code, `com.ibm.mqe.attributes.MQeWTLSCertAuthenticator`. There are some Java example authenticators, in the examples.attributes directory, which are based on user names and passwords. There is also a C example, WinCEAuthenticator, in the `examples\src\WinCEAuthenticator` directory. In addition to these, MQe allows you to write your own authenticator.

In queue-based security, authenticators are activated when a queue is first accessed and they can grant or deny access to the queue. When a queue is accessed from its local queue manager, the authenticator is activated when the first operation, for example put, get , or browse is performed on the queue. When a queue is accessed from a remote queue manager, MQe establishes a channel between the two queue managers and the authenticator is activated as part of establishing the channel.

Writing authenticators in Java:

All authenticators must extend the base authenticator class:

```
class MyAuthenticator extends com.ibm.mqe.MQeAuthenticator
```

The following methods in the base class can be overridden:

activateMaster()

The signature for this method is:

```
public byte[] activateMaster( boolean local ) throws Exception
```

It is invoked on the queue manager that initiates access to a queue. The parameter local indicates whether this is a local access; that is, the queue is on the same queue manager, local == true, or a remote access, local == false. The method should collect data to authenticate the queue manager or user and return the data in a byte array. The data is passed to the `activateSlave()` method. The `activateMaster()` method in the base class, MQeAuthenticator, simply returns null. It does not throw any exceptions. Any exceptions thrown by this method, in a subclass, are not caught by MQe itself, but are passed back to the user's code and terminate the attempt to access the queue.

activateSlave()

The signature for this method is:

```
public byte[] activateSlave( boolean local,  byte data[] ) throws Exception
```

This is invoked on the queue manager that owns the queue. The parameter local indicates whether this is a local access, i.e. initiated on the same queue manager, local == true, or a remote access, local == false. The parameter datacontains the data returned by the `activateMaster()` method. The `activateSlave()` method should validate this data. If it is satisfied with the data it should call the `setAuthenticatedID()` method to set the name of the authenticated entity, this indicates that the first stage of the authentication was successful. It can then collect data to authenticate the local queue manager and return it in a byte array. The data is passed to the `slaveResponse()` method. If it is not satisfied with the data, it throws an exception indicating the reason. The `activateSlave()` method in the base

class, MQeAuthenticator, checks whether the name of the authenticated entity has been set and if it has, it logs the name; it then returns null. It does not throw any exceptions. Any exceptions thrown by this method, in a subclass, are not caught by MQe itself, but are passed back to the initiating queue manager where they are re-thrown. MQe does not catch these exceptions on the initiating queue manager and they are passed back to the user's code and will terminate the attempt to access the queue.

`slaveResponse()`

The signature for this method is:

```
public void slaveResponse( boolean local, byte data[] ) throws Exception
```

It is invoked on the queue manager that initiates access to a queue. The local parameter indicates whether this is a local access, local == true, or a remote access, local == false. The parameter data contains the data returned by the activateSlave() method. If it is satisfied with the data it should call the `setAuthenticatedID()` method to set the name of the authenticated entity, this indicates that the second stage of the authentication was successful. If the `activateSlave()` method did not return any data, and the slaveResponse() method is satisfied with this, it still calls `setAuthenticatedID()` to indicate success. If it is not satisfied with the data, it throws an exception indicating the reason. The `slaveResponse()` method in the base class, MQeAuthenticator, simply returns null. It does not throw any exceptions. Any exceptions thrown by this method, in a subclass, are not caught by MQe itself, but are passed back to the user's code and terminate the attempt to access the queue.

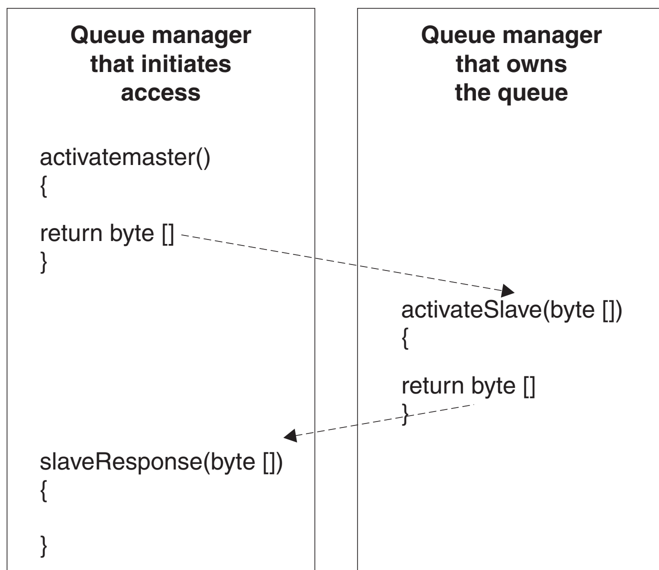


Figure 77. The slaveResponse() method in MQeAuthenticator

When a queue is accessed locally, the three methods are invoked in sequence on the local queue manager.

The example logon authenticator:

The example logon authenticator shows how to implement the three methods: `activateMaster()`, `activateSlave()`, and `slaveResponse()`.

It has a base class, `examples.attributes.LogonAuthenticator`, and three subclasses, one for the NTAuthenticator, one for the UnixAuthenticator, and one for the UseridAuthenticator. The base class provides common functionality and the subclasses provide functionality that is specific to the type of authenticator, that is NT, Unix, or Userid.

The `activateMaster()` method in the LogonAuthenticator class creates an empty MQeFields object and passes it into a method called `prompt()`. This is overridden in each of the subclasses, and in each case it displays a Java dialog box, collects data from it, masks the data with a simple exclusive OR operation, and adds the data to the MQeFields object. The exclusive OR is used in the example authenticators but in practice it does not provide much protection. The MQeFields object is dumped to provide a byte array which is returned by `activateMaster()`. The `activateMaster()` method is invoked on the queue manager that initiates access to the queue, so the dialog box is displayed by this queue manager.

```
public byte[] activateMaster(boolean local) throws Exception {
  MQeFields fields = new MQeFields();
/* for request fields       */
  this.prompt(fields);
/* put up the dialog prompt  */
  return (fields.dump());
/* return ID                 */
}
```

The `activateSlave()` method receives the data returned by `activateMaster()`, restores it into an MQeFields object and passes the object into the `validate()` method. This is overridden in each of the subclasses, and in each case it validates the data in a way appropriate to the authenticator. For example, in the NTAuthenticator subclass, the `validate()` method unmasks the data and passes it to the `logonUser()` method. This method uses Java Native Interface (JNI) to access the Windows security mechanism and check whether the user name and password are valid. If they are valid, the `validate()` method returns the user name, otherwise it throws an exception.

```
public byte[] activateSlave(boolean local,
                            byte    data[]) throws Exception {
  MQeFields fields = new MQeFields(data); /* work object              */
  try {
    authID = this.validate(fields);
  /* get the auth ID value    */
    setAuthenticatedID(authID);
  /* is it allowed ?          */
    super.activateSlave(local, data);
  /* call ancestor            */
    trace("_:Logon " + authID);
  /* trace                    */
    MQeFields result = new MQeFields();
  /* reply object             */
    result.putAscii(Authentic_ID, authID);/* send id                  */
    return (result.dump());
  /* send back as response    */
  }
  catch (Exception e) {
  /* error occured            */
    authID = null;
  /* make sure authID is null */
    setAuthenticatedID(null);
  /* invalidate               */
    throw e;
  /* re-throw the exception   */
  }
}
```

If the user name is valid, the `activateSlave()` method calls `setAuthenticatedID()` to register the user name and the calls `super.activateSlave()` which puts out a log message. It issues a trace message, adds the user name to an MQeFields object, dumps this to a byte array and returns it. If the user name is not valid, `validate()` throws an exception. The `activateSlave()` method catches the exception, ensures the authenticated id is null and re-throws the exception.

The `slaveResponse method()` receives the byte array returned by `activateSlave()` and restores it into an MQeFields object. The user name that was validated by `activateSlave()` is extracted from this and passed to `setAuthenticatedID()`.

```
public void slaveResponse(boolean local, byte data[])

throws Exception {   super.slaveResponse(local, data);       /* call ancestor*/
MQeFields fields = new MQeFields(data);                       /* work object*/
setAuthenticatedID(fields.getAscii(Authentic_ID));            /* id to check   */
}
```

These authenticators behave the same for both local and remote accesses, so they ignore the local parameter to these methods.

Writing authenticators in C:

In the C codebase, you need to provide at least four functions to implement an authenticator:

1. `new()`
2. `activateMaster()`
3. `ctivateSlave()`
4. `slaveResponse()`

In terms of functionality, functions 2 to 4 behave exactly the same as their Java counterpart implementation. If your `new()` function allocates any private memory, you then have to provide a `free()` function, which frees the private memory you have allocated.

new() The `new()` function is executed when the authenticator is loaded by MQe. It serves as an initialization function for the authenticator. Its main functionality includes:

- Allocating private memory, if required
- Notifying the MQe system of the implementations for the `activateMaster()`, `activateSlave()`, `slaveResponse()`, and `free()` functions
- Providing initial values for private variables

To notify the MQe of the existence of your implementation, call the `mqeClassAlias_add()` function, which has the following signature:

```
MQERETURN mqeClassAlias_add(MQERETURN * pExceptBlock,
                   MQeStringHndl hWinCEAuthName,
                            MQeStringHndl hModuleName,
                             MQeStringHndl hInitFuncName);
```

In the previous example, the `hWinCEAuthName` is a string name for the authenticator. The `hModuleName` is the dynamically loadable library file name in which your authenticator has been compiled into, and the `hInitFuncName` is the name of your new function, which can be an arbitrary name. The `new()`function has the following signature:

```
MQERETURN new(MQeAttrPlugin_SubclassInitInput * pInput,
                MQeAuthenticator_SubclassInitOutput * pOutput
          );
```

The pOutput points to an MQeAuthenticator_SubclassInitOutput structure, which needs to be filled in. The MQeAuthenticator_SubclassInitOutput contains the following fields:

MQEVERSION version;
Assign MQE_CURRENT_VERSION to this variable.

MQeStringHndl hClassName;
Assign the Java class name of the authenticator, MQeString, to this variable.

MQEBOOL regRequired;
Assign MQE_FALSE to this variable.

MQEKEYTYPE keyType;
Assign MQE_KEY_NULL to this variable.

MQeAuthenticator_FreeFunc fFree;
Assign the address of the free() function to this variable.

MQeAuthenticator_ActivateMasterPrepFunc fActivateMasterPrep;
Assign the address of the activateMaster() function to this variable.

MQeAuthenticator_ActivateSlavePrepFunc fActivateSlavePrep;
Assign the address of the activateSlave() function to this variable.

MQeAuthenticator_ProcessSlaveResponseFunc fProcessSlaveResponse;
Assign the address of the activateSlave() function to this variable.

MQeAuthenticator_CloseFunc fClose;
Assign NULL to this variable.

MQEVOID * pSubclassPrivateData;
Assign the address of authenticator's private data memory to this variable.

Any pointers or handles that are not used in the implementation must be initialised to NULL.

free() The signature of free() is:

```
MQERETURN free(MQeAuthenticatorHndl  hThis,
               MQeAttrPlugin_FreeInput * pInput,
               MQeAttrPlugin_FreeOutput * pOutput
              );
```

If the new() function allocates private memory, the pointer to the allocated memory can be retrieved into a pointer p using:

```
mqeAuthenticator_getPrivateData(hThis, pExceptBlock, (MQEVOID **) &p);
```

The pointer can then be used to free the memory. The MQeString assigned to the hClassName in the new() function, if any, are automatically freed by the system when mqeAttrBase_free is called.

activateMaster()
The signature of activateMaster() is:

```
MQERETURN activateMaster(MQeAuthenticatorHndl   hAuthenticator,
                          MQeAttrPlugin_ActivateMasterPrepInput *pInput,
                          MQeAttrPlugin_ActivateMasterPrepOutput * pOutput
                         );
```

Refer to description in the corresponding Java section for the required functionality for this function. The `pOutput` points to an `MQeAttrPlugin_ActivateMasterPrepOutput` structure which needs to be filled in. The `MQeAttrPlugin_ActivateMasterPrepOutput` contains the following fields:

`MQEINT32 * pOutputDataLen;`
: Assign the length of the output data for `activateSlave()` to this variable.

`MQEBYTE * pOutputData;`
: Assign the address of the output data buffer for `activateSlave()` to this variable.

activateSlave()

The signature of `activateSlave()` is:

```
MQERETURN activateSlave(MQeAuthenticatorHndl hAuthenticator,
                        MQeAttrPlugin_ActivateSlavePrepInput *pInput,
                         MQeAttrPlugin_ActivateSlavePrepOutput *pOutput
                      );
```

Refer to description in the corresponding Java section for the required functionality for this function. The pInput points to an `MQeAttrPlugin_ActivateSlavePrepInput` structure which contains the input from the `activateMaster()` and the `pOutput` points to an `MQeAttrPlugin_ActivateSlavePrepOutput` structure which needs to be filled in. The `MQeAttrPlugin_ActivateSlavePrepInput` contains the following fields:

`MQEINT32 * pInputDataLen;`
: Get the length of the input data from `activateMaster()` from this variable.

`MQEBYTE * pInputData;`
: Get the address of the input data buffer from `activateMaster()` from this variable.

The `MQeAttrPlugin_ActivateSlavePrepOutput` contains the following fields:

`MQEINT32 * pOutputDataLen;`
: Assign the length of the output data for `slaveResponse()` to this variable.

`MQEBYTE * pOutputData;`
: Assign the address of the output data buffer for slaveResponse() to this variable.

slaveResponse()

The signature of slaveResponse() is:

```
MQERETURN slaveResponse(MQeAuthenticatorHndl hAuthenticator,
                    MQeAttrPlugin_ProcessSlaveResponseInput *pInput,
                    MQeAttrPlugin_ProcessSlaveResponseOutput *pOutput
                     );
```

Refer to description in the corresponding Java section for the required functionality for this function. The `pInput` points to an

MQeAttrPlugin_ProcessSlaveResponseInput structure which contains the input from the activateSlave(). The MQeAttrPlugin_ProcessSlaveResponseInput contains the following fields:

MQEINT32 * pInputDataLen;
Get the length of the input data from activateSlave() from this variable.

MQEBYTE * pInputData;
Get the address of the input data buffer from activateSlave() from this variable.

The example WinCEAuthenticator:

The example WinCEAuthenticator shows how the methods listed in the previous section can be implemented. It is functionally very similar to the example NTAuthenticator in the Java code base.

Calling winCEAuthenticator_new() function implements the new() function. This allocates a private memory block to register the type of the authenticator, private to this implementation, filling-in private variables and the function pointers mentioned above so they point to the right function implementations, and set pOutput->hClassName to "WinCEAuthenticator". Notice that no "WinCEAuthenticator" is provided in the Java package. This is because the WinCEAuthenticator is designed to be executed only on a C client. The "WinCEAuthenticator" string is created for demonstration purposes only. The pOutput->hClassName must point to an existing Java class if the authenticator is to be used in a dialogue between a C client and a Java server.

```
 MQERETURN winCEAuthenticator_new(
         MQeAttrPlugin_SubclassInitInput * pInput,
         MQeAuthenticator_SubclassInitOutput * pOutput
        ) {

    MQeStringHndl hClassName;
    MQeExceptBlock * pExceptBlock =
              (MQeExceptBlock*) pOutput->pExceptBlock;

    (void)mqeString_newChar8(pExceptBlock,
                             &hClassName,
                             "WinCEAuthenticator");
    if (MQERETURN_OK == pExceptBlock->ec) {
     pOutput->pSubclassPrivateData = malloc(sizeof(MQEINT32));
     if (NULL != pOutput->pSubclassPrivateData) {
        *((MQEINT32 *)pOutput->pSubclassPrivateData) = AUTHENTICATOR;
        pOutput->hClassName = hClassName;
        pOutput->regRequired = MQE_FALSE;
        /* key type unknown */
        pOutput->keyType = MQE_KEY_NULL;

        /* pointers to subclass implementations of support methods */
        pOutput->fFree = winCEAuthenticator_free;
        pOutput->fActivateMasterPrep = winCEAuthenticator_activateMasterPrep;
        pOutput->fActivateSlavePrep = winCEAuthenticator_activateSlavePrep;
        pOutput->fProcessSlaveResponse = winCEAuthenticator_processSlaveResponse;
        pOutput->fClose = NULL;
     } else {
        pExceptBlock->ec  = MQERETURN_ALLOCATION_FAIL;
        pExceptBlock->erc = MQEREASON_NA;
     }
   }

   return pExceptBlock->ec;
}
```

Calling `winCEAuthenticator_free()` implements the `free()` function. It retrieves the private memory block allocated by `winCEAuthenticator_new()`, making sure the authenticator has got the right private signature, and then frees the memory block.

```
MQERETURN winCEAuthenticator_free(MQeAuthenticatorHndl  hThis,
                              MQeAttrPlugin_FreeInput * pInput,
                              MQeAttrPlugin_FreeOutput * pOutput
                                  )  {

   MQeExceptBlock * pExceptBlock = (MQeExceptBlock*)
                     pOutput->pExceptBlock;
   MQEINT32 * pType;

   pExceptBlock->ec  = MQERETURN_INVALID_ARGUMENT;
   pExceptBlock->erc = MQEREASON_INVALID_SIGNATURE;

   if ((NULL != hThis) &&
       (MQERETURN_OK == mqeAuthenticator_getPrivateData(hThis,
                                                pExceptBlock,
                                                (MQEVOID**)  &pType))
                                                    ) {
      /* make sure it is an authenticator created here */
      if (AUTHENTICATOR == *pType) {
         pExceptBlock->ec  = MQERETURN_OK;
         pExceptBlock->erc = MQEREASON_NA;
         free(pType);
      }
   }

   return pExceptBlock->ec;
}
```

Calling `winCEAuthenticator_activateMasterPrep()` implements the `activateMaster()` function. It creates an empty MQeFields structure and passes it into a function called `prompt()`. The `prompt()` function:

- Displays a dialogue box
- Collects data from the dialogue box
- Masks the data with a simple exclusive OR operation
- Adds the data to the MQeFields object

The exclusive OR is used in the example authenticators, but in practice it does not provide much protection. The MQeFields structure is then dumped to provide a byte array, which is returned by `winCEAuthenticator_activateMasterPrep()`.

```
MQERETURN winCEAuthenticator_activateMasterPrep(
    MQeAuthenticatorHndl  hAuthenticator,
    MQeAttrPlugin_ActivateMasterPrepInput *  pInput,
    MQeAttrPlugin_ActivateMasterPrepOutput * pOutput) {

   static MQeFieldsHndl hActivateMasterFields = NULL;
   MQEINT32 * pOutputDataLen = pOutput->pOutputDataLen;
   MQEBYTE * pOutputData = pOutput->pOutputData;
   MQeExceptBlock * pExceptBlock =
            (MQeExceptBlock*) pOutput->pExceptBlock;


   /* initialize exception block */
   pExceptBlock->ec  = MQERETURN_OK;
   pExceptBlock->erc = MQEREASON_NA;

   if (NULL == hActivateMasterFields) {
      /* get data for authentication */
      (void)mqeFields_new(pExceptBlock,
               &hActivateMasterFields);
```

```
      if (MQERETURN_OK == pExceptBlock->ec) {
         /**
          * Write your code here which puts the input data,
          * for example., userid, password into hActivateMasterFields.
          * The format is not important as long as it can be
          * understood by your corresponding code in
          * winCEAuthenticator_activateSlavePrep, which digests
          * these data.
          */
     prompt(hActivateMasterFields, pExceptBlock);}
    }

    if (MQERETURN_OK == pExceptBlock->ec) {
       /* dump the fields */
       (void)mqeFields_dump(hActivateMasterFields,
                            pExceptBlock,
                            pOutputData,
                            pOutputDataLen);
    }

    if ((NULL != hActivateMasterFields) &&
        ((NULL != pOutputData) ||
            (MQERETURN_OK != pExceptBlock->ec))) {
       /**
        * Caller has supplied a buffer or operation failed.
        * No need to keep the Fields any more.
        */
       (void)mqeFields_free(hActivateMasterFields, NULL);
       hActivateMasterFields = NULL;
    }

    return pExceptBlock->ec;
 }
```

The `winCEAuthenticator_activateSlavePrep()` implements the `activateSlave()` function. The `winCEAuthenticator_activateSlavePrep()` method receives the data returned by `winCEAuthenticator_activateMasterPrep()`, restores it into an MQeFields structure and passes it into a `validate()` function. The `validate()` function unmasks the data and passes it to the system `LogonUser()` function. This function checks if the user name and password are valid.

On a WinCE system, the `LogonUser()` never returns if the user name and password are not valid. The following `winCEAuthenticator_activateSlavePrep()` and `winCEAuthenticator_processSlaveResponse()` implementations, however, assume that LogonUser() will always return with a value indicating whether or not the input is valid, in order to demonstrate what you need to do. If the user name and password are valid, the `winCEAuthenticator_activateSlavePrep()` function calls `mqeAuthenticator_setAuthenticatedID()` to register the user name as if the code is running on a server. It may be that this code is running on a client just as the `winCEAuthenticator_activateMasterPrep`. It then adds the user name to an MQeFields, dumps this to a byte array, and returns it. If the user name is not valid, the `winCEAuthenticator_activateSlavePrep()` function returns an error.

```
 MQERETURN winCEAuthenticator_activateSlavePrep(
                MQeAuthenticatorHndl  hAuthenticator,
                MQeAttrPlugin_ActivateSlavePrepInput *  pInput,
                MQeAttrPlugin_ActivateSlavePrepOutput * pOutput) {

    static MQeFieldsHndl hActivateSlaveFields = NULL;
    MQeFieldsHndl hTempFields = NULL;
    MQEINT32 inputDataLen = pInput->inputDataLen;
    MQEBYTE * pInputData = pInput->pInputData;
    MQEINT32 * pOutputDataLen = pOutput->pOutputDataLen;
    MQEBYTE * pOutputData = pOutput->pOutputData;
```

```
   MQeExceptBlock * pExceptBlock =
              (MQeExceptBlock*) pOutput->pExceptBlock;

   /* initialize exception block */
   pExceptBlock->ec  = MQERETURN_OK;
   pExceptBlock->erc = MQEREASON_NA;

   if (NULL == hActivateSlaveFields) {
/* restore input */
      (void)mqeFields_new(pExceptBlock, &hTempFields);
      if (MQERETURN_OK == pExceptBlock->ec) {
           /* restore it into an MQeFields */
           (void)mqeFields_restore(hTempFields,
                                   pExceptBlock,
                                   pInputData,
                                   inputDataLen);
         if (MQERETURN_OK == pExceptBlock->ec) {
            MQeStringHndl hAuthenticID = NULL;

            /**
             * put your code, which digests(authenticates)
             * the input data your gathered in the
             * winCEAuthenticator_activateMasterPrep().
             * If successful, create an AuthenicateID string
             * in hAuthenticID.
             */
            (void)validate(hTempFields, pExceptBlock,
                    &hAuthenticID);

              if (MQERETURN_OK == pExceptBlock->ec) {
                 /**
                * If successfully authenticated,
                * set local id variable (recored a success)
                */
               (void)mqeAuthenticator_setAuthenticatedID(hAuthenticator,
                                                        pExceptBlock,
                                                            hAuthenticID);
               /* preparation for sending the id to the master */
               if (MQERETURN_OK == pExceptBlock->ec) {
                  /**
                   * Send the hAuthenticID to the Master,
                   * indicating a a success.
                   */
                  (void)mqeFields_new(pExceptBlock,
                          &hActivateSlaceFields);
                if (MQERETURN_OK == pExceptBlock->ec) {
                   MQeStringHndl hAuthenticIDField;
            (void)mqeString_newChar8(pExceptBlock,
                                      &hAuthenticIDField,
                                      AUTHENTIC_ID);
                     if (MQERETURN_OK == pExceptBlock->ec) {
                        (void)mqeFields_putAscii(hActivateSlaveFields,
                                                 pExceptBlock,
                                                 hAuthenticIDField,
                                                 hAuthenticID);
                        (void)mqeString_free(hAuthenticIDField, NULL);
                     }
                  }
               }
            }
         }
         (void)mqeFields_free(hTempFields, NULL);
      }
   }

   if (MQERETURN_OK == pExceptBlock->ec) {
      /* dump the fields */
```

```
      (void)mqeFields_dump(hActivateSlaveFields,
                           pExceptBlock,
                           pOutputData,
                         pOutputDataLen);
   }

   if ((NULL != hActivateSlaveFields) &&
         ((NULL != pOutputData) || (MQERETURN_OK != pExceptBlock->ec))) {
        /**
         * Caller has supplied a buffer or operation failed.
         * No need to keep the Fields any more.
         */
        (void)mqeFields_free(hActivateSlaveFields, NULL);
        hActivateSlaveFields = NULL;
   }

   return pExceptBlock->ec;
}
```

Calling `winCEAuthenticator_processSlaveResponse()` implements the `slaveResponse()` function. The `winCEAuthenticator_processSlaveRespons()` function receives the byte array returned by `winCEAuthenticator_activateSlavePrep()` and restores it into an MQeFields structure. The user name, validated by `activateSlave()`, is extracted from this and passed to `mqeAuthenticator_setAuthenticatedID()`.

```
MQERETURN winCEAuthenticator_processSlaveResponse(
         MQeAuthenticatorHndl     hAuthenticator,
         MQeAttrPlugin_ProcessSlaveResponseInput *   pInput,
         MQeAttrPlugin_ProcessSlaveResponseOutput *   pOutput
        ) {

   MQEINT32 inputDataLen = pInput->inputDataLen;
   MQEBYTE * pInputData = pInput->pInputData;
   MQeFieldsHndl hFields;
   MQeExceptBlock * pExceptBlock =
                (MQeExceptBlock *)pOutput->pExceptBlock;

   /* initialize exception block */
   pExceptBlock->ec  = MQERETURN_OK;
   pExceptBlock->erc = MQEREASON_NA;

   /* restore input */
   (void)mqeFields_new(pExceptBlock, &hFields);
   if (MQERETURN_OK == pExceptBlock->ec) {
      (void)mqeFields_restore(hFields,
                             pExceptBlock,
                             pInputData,
                             inputDataLen);
      /* get ID */
      if (MQERETURN_OK == pExceptBlock->ec) {
         MQeStringHndl hAuthenticIDField;

         (void)mqeString_newChar8(pExceptBlock,
                                 &hAuthenticIDField,
                                 AUTHENTIC_ID);
         if (MQERETURN_OK == pExceptBlock->ec) {
            MQeStringHndl hAuthenticID;
      (void)mqeFields_getAscii(hFields,
                               pExceptBlock,
                               &hAuthenticID,
                               hAuthenticIDField);
            /** If the above call failed,
             * then the authentication by the slave was not successful.
             */
            if (MQERETURN_OK == pExceptBlock->ec) {
```

```
                /* set local ID */
                (void)mqeAuthenticator_setAuthenticatedID(hAuthenticator,
                                                          pExceptBlock,
                                                          hAuthenticID);
            }
            (void)mqeString_free(hAuthenticIDField, NULL);
        }
    }
    (void)mqeFields_free(hFields, NULL);
  }
  return pExceptBlock->ec;
}
```

Queue manager based security

Security features can be added at the queue-manager level by configuring the queue manager and its private registry.

Configuring queue manager security:

This section shows how to configure a queue manager and a private registry with security features.

Setting up the queue manager:

In order to configure a queue manager's private registry, which can be shared by its' queues, do the following:

1. When starting the queue manager, present the private registry logon PIN. If autoregistration with a mini-certificate server is required, the CertReqPIN, KeyRingPassword, and CAIPAddrPort parameters must also be presented, on opening the registry.
2. The mini-certificate server is running if autoregistration is required.

Setting up a private registry:

A private registry is relevant only if one of the queue-attribute properties prerequisites it. In order to establish a queue manager private registry, which can be shared by its' queues, the following conditions must be met:

1. The owning queue manager must itself have a registry of type private registry.
2. The owning queue manager must have previously auto-registered with the mini-certificate server. This must have been primed to allow queue registry before the queue private registry can be established. if auto registration with a mini-certificate server is required.
3. In starting the queue manager, the queue manager private registry logon PIN, CertReqPIN, KeyRingPassword, and CAIPAddrPort were passed whilst opening the registry. If a CertReqPIN different from the queue manager's is used for the queue, it is currently necessary to first shutdown the owning queue manager, replace the original CertReqPIN with the new one, and then start the queue manager again. Auto-registration will then be triggered using the new CertReqPIN when the queue private registry is activated first time.
4. The mini-certificate server is running, if autoregistration with the mini-certificate server is required.

If queue private registry (instead of the queue manager's) is required, for example, the target registry property of the queue has been set to "Queue" for com.ibm.mqe.attributes.MQeWTLSCertAuthenticator.

Due to the intensity of numerical computation involved, auto-registration may take 10-20 minutes on a handheld device.

Security configuration example:

Security attribute properties can be added to a queue using the com.ibm.mqe.administration.MQeQueueAdminMsg class and its subclasses. The security attribute properties are defined as parameters of the administration message. The following example (examples.security.createSecureQueue) creates a new queue on an existing client queue manager. It creates the queue with a cryptor, compressor, authenticator, and attribute rule. It is not necessary to add all of these attributes and any of them could be omitted. A cryptor on a local queue uses a key seed based on the queue manager private registry logon PIN. Therefore, it is important to present the right PIN when starting the queue manager.

The example starts with a class header:

```
package examples.security;

import java.io.File;
import com.ibm.mqe.*;
import com.ibm.mqe.administration.*;
import examples.queuemanager.MQePrivateClient;

/** createSecureQueue.java
 * <p>This creates a secure queue on an existing queue manager. The queue is
 * created with an authenticator, cryptor, compressor and attribute rule.
 * The queue manager must have a private registry, so that the queue can be
 * given a cryptor.
 *
 * <p>The program requires two command line parameters.
 *
 * <p>The first parameter is a configuration file for the queue manager. This
 * is used to start the queue manager as a client.
 *
 * <p>The second parameter is the PIN for the queue manager's private
 * registry.
 *
 **/

public class createSecureQueue
{
```

First you define the name of the queue you want to add:

```
  // the name of the queue
  String qName = "protQueue";
```

The attributes are defined by their class names:

```
  // define the attributes we want the queue to have. These are defined by
  // their class names.
  String cryptorType       = "com.ibm.mqe.attributes.MQeDESCryptor";
  String compressorType    = "com.ibm.mqe.attributes.MQeGZIPCompressor";
  String authenticatorType = "examples.attributes.NTAuthenticator";
  String attributeRule     = "com.ibm.mqe.MQeAttributeRule";
```

They are followed by some definitions of local variables:

```
  //local variables
  MQePrivateClient client;
  MQeQueueManager  clientQM;
  String           clientQMName;
  MQeQueueAdminMsg msg;
```

The example adds the queue directly to the local queue manager, so the queue manager must be activated:

```
/**
 * open the queue manager as a client
 *
 * @param configFile the configuration (.ini) file for the queue manager
 * @param qmPIN      the PIN for the queue manager's registry
 * @exception  java.lang.Exception  propagated from invoked methods
 **/
void openQM(String configFile, String qmPIN) throws Exception
{
  // start the queue manager as a client
  client = new MQePrivateClient(configFile, qmPIN, null, null);

  //save the queue manager and its name
  clientQM = client.queueManager;
  clientQMName = clientQM.getName();
}
```

The MQeQueueAdminMsg is created and values added to it as normal. A correlation id is added to the message to make it easy to find the reply message. All the security attributes are added as parameters to the message, that is, they are added to a separate MQeFields object which is passed to the msg.create(parms) method:

```
/**
 * create the admin message to add the queue attributes
 *
 * @exception  java.lang.Exception  propagated from invoked methods
 **/
void createAdminMsg() throws Exception
{
  // the file descriptor
  String FileDesc = "MsgLog:.";

  // create an Admin msg to add the queue
  msg = new MQeQueueAdminMsg();
  msg.setTargetQMgr(clientQMName);
  msg.setName(clientQMName, qName);
  msg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);
  msg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);
  msg.putAscii(MQe.Msg_ReplyToQMgr, clientQMName);
  msg.putArrayOfByte(MQe.Msg_CorrelID,
      Long.toHexString(clientQM. uniqueValue()).getBytes());

  // define parameter values for the queue
  MQeFields parms = new MQeFields();
  parms.putUnicode(msg.Queue_Description, "DES protected queue");
  parms.putAscii(msg.Queue_FileDesc, FileDesc );

  // this is where we specify the queue attributes
  parms.putAscii(msg.Queue_Cryptor, cryptorType);
  parms.putAscii(msg.Queue_Compressor, compressorType);
  parms.putAscii(msg.Queue_Authenticator, authenticatorType);
  parms.putAscii(msg.Queue_AttrRule, attributeRule);

  //add the parameters to the message
  msg.create(parms);
}
```

The message is sent to the Admin Queue on the local queue manager:

```
/**
 * send the admin message to the client queue manager
 *
 * @exception  java.lang.Exception  propagated from invoked methods
```

```
   **/
  void sendAdminMsg() throws Exception
{
    // send the Admin msg
    System.out.println("putting Admin Msg to QM/queue:" +
    clientQMName + "/" + MQe.Admin_Queue_Name);
    clientQM.putMessage(clientQMName, MQe.Admin_Queue_Name, msg, null, 0);
  }
```

The correlation id is used in a filter to find the correct reply. The example waits up to 3 seconds for the reply:

```
  /**
   * wait for a reply message and process it to determine success or failure
   *
   * @exception  java.lang.Exception  propagated from invoked methods
   **/
  void processReply() throws Exception
    {
    // use the CorrelID to create a filter for the reply message
    MQeFields replyFilter = new MQeFields();
    replyFilter.putArrayOfByte(MQe.Msg_CorrelID,
     msg.getArrayOfByte(MQe.Msg_CorrelID));

    // get the Admin Reply msg
    MQeMsgObject reply = clientQM.waitForMessage(clientQMName,
                MQe.Admin_Reply_Queue_Name,
                replyFilter,
                null,
                0,
3000);
    if (reply instanceof MQeAdminMsg)
    {
      MQeAdminMsg adminReply = (MQeAdminMsg)reply;
      System.out.println("Admin Reply Msg received");
      if (adminReply.getRC() == MQeAdminMsg.RC_Success)
        System.out.println("Queue added OK");
      else
System.out.println("create Queue failed:" +
    adminReply.getReason());
    }
    else
      System.out.println("reply message is not an admin message");
  }
```

The queue manager needs to be closed:

```
  /**
   * close the queue manager
   *
   * @exception  java.lang.Exception  propagated from invoked method
   **/
  void close() throws Exception
  {
    clientQM.close();
  }
```

The main() method for the example is:

```
  /**
   * main method.
   *
   * @param  args  The command line arguments. The first is a configuration
   *              (.ini) file for the queue manager, the second is the PIN
   *              for the queue manager's private registry.
   *
   **/
  public static void main(String [] args)
```

```
  {
    createSecureQueue secQueue = new createSecureQueue();
    // check the command line arguments
    if (args.length < 2)
      System.err.println("usage: createSecureQueue configFile qmPIN");
    else
    {
      try
      {
        secQueue.openQM(args[0], args[1]);
        secQueue.createAdminMsg();
        secQueue.sendAdminMsg();
        secQueue.processReply();
        secQueue.close();
      }
      catch (Exception e)
      {
        System.out.println("Exception caught:" + e);
      }
    }
  }
}
```

Attribute rules can also be set on channels using the ChannelAttrRules keyword in the configuration file used at queue manager creation time. MQe defaults the keyword to com.ibm.mqe.MQeAttrubuteRule.

Channel level security

When data is sent between a queue manager and a remote queue, the queue manager opens a channel to the remote queue manager that owns the queue. By default, if the remote queue is protected, for example with a cryptor, the channel is given exactly the same level of protection as the queue. For efficiency in queue-based security, an MQe channel uses symmetric cryptors (for example, DES, 3DES, MARS, RC4, RC6); a consequence of which is that the two queue managers at either end must use the same encryption key. When such a channel is established, a protocol, called the Diffie Hellman key exchange, is used to establish a secret key that only the two queue managers know.

This protocol is susceptible to a ″man in the middle″ attack, but for that to be successful, the ″man in the middle″ must know some of the data that is fed into the Diffie Hellman protocol. This data is held in the com.ibm.mqe.attributes.MQeDHk class. It is possible for an attacker to get hold of this data, by examining the shipped MQe classes. However, this data can be changed by running the com.ibm.mqe.attributes.MQeGenDH utility; it generates a new Java source file com.ibm.mqe.attributes.MQeDHk.java. This file can then be compiled into a replacement com.ibm.mqe.attributes.MQeDHk.class file.

When the com.ibm.mqe.attributes.MQeWTLSCertAuthenticator is used, the two queue managers (or queues) swap certificates in order to authenticate each other. If this is used in conjunction with a cryptor on the queue, the exchanges which establish the secret key for the cryptor are protected with the public keys from the certificates, making a ″man in the middle″ attack even more difficult.

With synchronous remote queues, queue-based security is relatively simple. In this case a message is put to a synchronous remote queue definition that has the same security attributes as the destination queue. The message is transmitted over a channel with appropriate security attributes and is stored on the secure queue.

With asynchronous remote queues, especially Store-and-forward queues and Home-server queues, the transmitting and receiving queues are more likely to have different security attributes. These differences have to be managed during message transfer. Once a message has been put to an asynchronous queue it is transmitted from one queue to another until it reaches its destination. A queue manager is responsible for requesting the transfer of the message between a pair of queues and another queue manager is responsible for responding to the request. If queue based security is used, the requesting queue manager establishes a channel with security attributes that match the queue that it owns. The queue manager receiving the request checks that the channel attributes are sufficient for its queue.

For example, suppose a client queue manager has a queue with a DES cryptor on it and messages are routed from this to a server's Store-and-forward queue that has a MARS cryptor. When the client is triggered to send a message it establishes a DES encrypted channel to the server; the server asks the Store-and-forward queue whether it will accept messages over a DES encrypted channel. If the Store-and-forward queue considers DES is not as strong as its own MARS cryptor (determined by the queue attribute rule), it would throw an ″attribute mismatch″ exception.

A Home-server queue trying to pull messages from a Store-and-forward queue needs a cryptor that is at least as strong as that on the Store-and-forward queue, because the Home-server queue is at the initiating end of the request. Once the Home-server queue has received the message it can store it on a local queue that has any level of protection. This behavior can be changed by using different attribute rules on the queues. For example, if the attribute rule always allows reuse, the queue will accept channels with any cryptor.

Trying to send a message from a queue with a weaker cryptor to a queue with a stronger cryptor usually results in an ″attribute mismatch″ exception. However if a channel with a strong cryptor already exists between the queue managers, this can be reused (depending on the attribute rules on the channel) and result in the message being delivered.

One slight exception to the above behavior is when a Store-and-forward queue is used to forward (push) messages to other queues. The Store-and-forward queue establishes a channel with security attributes that match its own. However, in this case the destination queue accepts the channel without checking its attributes against the queue's. For example, a Store-and-forward queue without a cryptor would establish a channel without a cryptor and this would be used to forward messages to a destination queue even if the queue had a cryptor on it. Normally, with other queue types, this would result in an ″attribute mismatch″ exception. When using a Store-and-forward queue in this way, you should ensure that it has a cryptor that is comparable to any cryptor on a destination queue. This does not apply when a Home-server queue polls for messages from a Store-and-forward queue (in this case the Home-server queue establishes the channel, not the Store-and-forward queue).

Channel attribute rules:

To reduce the number of channels open concurrently, the queue manager can reuse an existing channel if its level of protection is adequate. If none of the channels has a suitable level of protection, the queue manager can also change (upgrade) the level of protection on an existing channel to match that required for the queue. This kind of behavior is governed by the MQeattributeRule on both the queue and the channel. These rules apply to the attribute on the queue (and channel), they are

not the same as queue rules. Attribute rules are set on a queue when it is created or modified using administration messages.

The isAcceptable() method on the MQeAttributeRule class determines if a channel can be reused. This provides protection against inconsistency in the queue attribute rules on the local and target queue managers. If the isAcceptable() method returns true, the channel is used. Otherwise, the channel will not be reused.

If none of the existing channels can be reused, the queue manager checks if any of the channels can be upgraded to the required level. The permit() method on the MQeAttributeRule class determines this. If the permit() method returns true, the channel is upgraded. Otherwise, the channel is upgraded.

MQe provides a default rule, com.ibm.mqe.MQeAttributeRule (identical to examples.rules.AttributeRule. This is specified as the attribute rule for a queue by MQe by default.

Note: This is different from setting attribute rule to null.

This rule allows a channel to be used for a queue if the following conditions are met:

1. If the queue has an authenticator, the channel must have the same type of authenticator. If the queue does not have an authenticator, it does not matter whether the channel has one or not.
2. If the queue has a cryptor, the channel must have a cryptor that is the same type as or better than that on the queue. If the queue does not have a cryptor it does not matter whether the channel has one or not. Here ″better″ is defined as:
   - Any cryptor is the same as or better than XOR.
   - Any cryptor, except XOR, is the same as or better then DES.
   - The remaining cryptors (Triple DES, RC4, RC6, and MARS) are considered equal to each other and all better than XOR and DES.
3. It does not matter what compressors are defined for the queue or channel.

This rule has the following upgrade behavior:

1. If the channel has been authenticated it cannot be upgraded, but if it does not have one, an authenticator can be added to a channel.
2. A cryptor can be added to a channel or strengthened (using the criteria for ″better″ described above). A cryptor cannot be removed from the channel or replaced with a weaker cryptor.
3. A compressor can be changed, added to, or removed from the channel.

If the attribute rule is explicitly set to null, MQe adopts an internal rule, com.ibm.mqe.MQeAttributeDefaultRule. This rule only accepts a channel that has exactly the same authenticator (and authenticated to the same entity), cryptor, and compressor as itself for reuse and always allow channel upgrade.

Because of the way channel security works, when a specific attribute rule is specified for a target queue, it forces the local queue manager to create an instance of the same attribute rule (examples.rules.AttributeRule and com.ibm.mqe.MQeAttributeRule are treated as the same rule for backward compatibility). A null rule can be specified for the target queue, to avoid the need to have the same attribute rule available remotely.

While the com.ibm.mqe.MQeAttributeRule provides practical defaults, there may be a solution-specific reason why different behavior is required. You can modify the way channels are reused by extending or replacing the default com.ibm.mqe.MQeAttributeRule with rules that define the desired behavior.

Certificate management

MQe can use private or public key encryption for message level security using the MQeMTrustAttribute, and for queue based security using the MQeWTLSCertAuthenticator. Any entity, for example queue manager, queue, application, person, which needs private and public keys must have a private registry. When the registry is initialized it generates and stores the keys, if the associated information is supplied.

The private key is encrypted and stored directly in the registry. The public key is sent to the certificate server, which returns a public certificate containing the public key, and the registry stores the certificate. For message level security, the certificates must also be copied to public registries so that they are available to other entities that need them. This is not required for queue based security.

The certificate server normally issues certificates, which are valid for 12 months. The certificates cannot be used once they have expired, so it is important to keep track of the expiry dates and to renew the certificates before they expire.

Examining certificates:

Certificates can be examined using the `com.ibm.mqe.attributes.MQeListCertificates` class. This class opens a registry and allows you to list all the certificates in it, or to examine specific certificates by name. To use the class, you must supply the name of the registry and an MQeFields object that contains the information required to open it:

`MQeRegistry.LocalRegType (ascii)`
For a public registry, set this parameter to `com.ibm.mqe.registry.MQeFileSession`. For a private registry, set it to `com.ibm.mqe.registry.MQePrivateSession`.

`MQeRegistry.DirName (ascii)`
The name of the directory holding the registry files.

`MQeRegistry.PIN(ascii)`
The PIN protecting the registry. This is only required for private registries.

No other parameters are required to open the registry for this class. If the registry is a public registry with the name ″MQeNode_PublicRegistry″and the class is initialised in the directory that contains the registry, the MQeFields object can be null. If the registry belongs to the mini-certificate server, its name is ″MiniCertificateServer″. If the registry belongs to a queue, its name is ″MiniCertificateServer″.

```
MQeListCertificates list;
String fileRegistry = "com.ibm.mqe.registry.MQeFileSession";
String privateRegistry = "com.ibm.mqe.registry.MQePrivateSession";

void open(String regName, String regDirectory,
      String regPIN) throws Exception
{
    MQeFields regParams = new MQeFields();
    // if regPIN == null, assume file registry
    String regType = (regPIN == null) ?
```

```
                fileRegistry : privateRegistry;
        regParams.putAscii(MQeRegistry.RegType, regType);
        regParams.putAscii(MQeRegistry.DirName, regDirectory);
        if (regPIN != null)
            regParams.putAscii(MQeRegistry.PIN, regPIN);

        list = new MQeListCertificates(regName, regParams);
}
```

This constructor opens the registry. Once this has been done, the registry entries for the certificates can be retrieved. They can be retrieved either individually by name:

```
MQeFields entry = list.readEntry(certificateName);
```

or all the certificate entries in the registry can be retrieved together:

```
MQeFields entries = list.readAllEntries();
```

The value returned from `readAllEntries()` is an MQeFields object that contains a field for each certificate in the registry, the name of the field is the name of the certificate and the contents of the field is an MQeFields object containing the registry entry. You can process each registry entry using an enumeration:

```
     Enumeration enum = entries.fields();

            if (!enum.hasMoreElements())
                System.out.println("no certificates found");
            else
            {
                while (enum.hasMoreElements())
                {
                    // get the name of the certificate
                    String entity = (String) enum.nextElement();
                    // get the certificate's registry entry
                    MQeFields entry = entries.getFields(entity);

                    // do something with it
                    ...
                }
            }
```

The certificate can be obtained from the registry entry using the `getWTLSCertificate()` method:

```
   Object certificate = list.getWTLSCertificate(entry);
```

Information can now be obtained from the certificate:

```
   String subject   = list.getSubject(certificate);
  String issuer    = list.getIssuer(certificate);
  long   notBefore = list.getNotBefore(certificate);
  long   notAfter  = list.getNotAfter(certificate);
```

The `notBefore` and `notAfter` times are the number of seconds since the midnight starting 1st January 1970, that is the standard UNIX format for dates and times.

Finally, the list object should be closed:

```
list.close();
```

The MQeListCertificates class is used in the example program, `examples.certificates.ListWTLSCertificates`, which is a command-line program that lists certificates.

The program has one compulsory and three optional parameters:

ListWTLSCertificates <regName>[<ini file>][<level>][<cert names>]

where:

regName
: The name of the registry whose certificates are to be listed. It can be a private registry belonging to a queue manager, a queue or another entity. It can be a public registry, or, for the administrator, it can be the mini-certificate server's registry. If you want to list the certificates in a queue's registry, you must specify its name as `<queue manager>+<queue>`, for example `myQM+myQueue`. If you want to list the certificates in a public registry, it must have the name `MQeNode_PublicRegistry`. It will not work for a public registry with any other name. The name of the mini-certificate server's registry is MiniCertificateServer .

ini file
: This is the name of a configuration file that contains a section for the registry. This is typically the same configuration file that is used for the queue manager or mini-certificate server. For a queue, this is typically the configuration file for the queue manager that owns the queue. This parameter should be specified for all registries except public registries, for which it can be omitted.

level
: The level of detail for the listing. This can be:

- ″-b″ or ″-brief″, which prints the names of the certificate, one name per line.
- ″-f″ or ″-full″, which prints the names of the certificates and some of the contents.

This parameter is optional and if omitted the ″brief″ level of detail is used.

cert names
: This is a list of names of the certificates to be listed. It starts with the flag ″`-cn`″ followed by names of the certificates, for example `-cn ExampleQM putQM` .If this parameter is used, only the named certificates are listed. If this parameter is omitted, all the certificates in the registry are listed.

The MQe_Explorer configuration tool can also be used to examine certificates which belong to queue managers or queues.

Renewing certificates:

To ensure continuity of service, you are advised to renew certificates before they expire. Certificates are renewed using the same mini-certificate issuance service that originally issued them. Before requesting a renewal, the request must be authorized with the issuance service and a one-time-use certificate request PIN obtained, in just the same way as for the initial certificate issuance.

When a certificate is renewed, the new certificate contains the same public key as the old certificate. For additional security, you may wish to change credentials regularly. This involves generating a new private and public key, storing the new private key in the registry, and requesting a new certificate for the public key. If you use message level security with the MTrustAttribute, and change credentials, you will not be able to use the new credentials to read messages sent with the old credentials. The old credentials are not deleted, but are renamed within the registry so that they are still available.

The class `com.ibm.mqe.registry.MQePrivateRegistryConfigure` can be used both to renew certificates and to generate new credentials. To use the class, you must supply the name of the registry, an MQeFields object that contains the information required to open it, and optionally the registry's PIN.

Security services

MQe provides the following services to assist with security:

Private registry services
MQe private registry provides a repository in which public and private objects can be stored. It provides (login) PIN protected access so that access to a private registry is restricted to the authorized user. It also provides additional services so that functions can use the entity's private key, (for digital signature, and RSA decryption) without the private credentials leaving the PrivateRegistry instance.

These services are used by queue-based security and message-level security using MQeTrustAttribute.

Public registry services
MQe public registry provides a publicly accessible repository for mini-certificates.

These services can be used by queue-based and message-level security.

Mini-certificate issuance service
MQe provides SupportPac ES03, ″MQe WTLS Mini-Certificate Server″, which includes a default *mini-certificate issuance service* which you can configure to issue mini-certificates to a carefully controlled set of entity names.

These services can be used by queue-based and message-level security.

Private registry service

This topic describes the private registry service provided by MQe. Note that the private registry service applies only to the Java codebase.

Private registries:

Some security properties, such as com.ibm.mqe.attributes.MQeWTLSCertAuthenticator, prerequisite an appropriate private registry where the entity's private/public keys can be found, and, in some cases, the queue manager's public registry where foreign entities' public keys can be found. This happens when a security attribute uses a public/private key based algorithm to perform encryption/authentication.

There are two types of private registries, queue manager owned and queue owned, and each private registry only stores its owner's security credentials. The queue manager's credential, however, can be shared by the queues it owes. For this reason, if the com.ibm.mqe.attributes.MQeWTLSCertAuthenticator class authenticator is used, an additional parameter ″target registry″ on the queue attribute that the authenticator is attached to must also be set. This parameter determines which registry is to supply the credentials for authentication, and can have the value of either ″Queue manager″ or ″Queue″.

If ″Queue manager″ is specified, the credentials used are those of the queue manager owning the queue, and come from the private registry of the queue

manager. The queue manager originally obtains these credentials through auto-registration with the mini-certificate server. This option is the recommended default.

If ″Queue″ is specified, the credentials used are those of the queue itself, and come from the private registry of the queue. The queue originally obtains these credentials through auto-registration with the mini-certificate server as well.

See “Mini-certificate issuance service” on page 241 for issues related to mini-certificate management.

Private registry usage guide:

Prior to using queue-based security, MQe-owned authenticatable entities must have credentials. This is achieved by completing the correct configuration so that auto-registration of queue managers is triggered. This requires the following steps:

1. Setup and start an instance of MQe mini-certificate issuance service.
2. Using MQe_MiniCertificateServer, add the name of the queue manager as a valid authenticatable entity, and the entity’s one-time-use certificate request PIN.
3. Configure MQePrivateClient1.ini and MQePrivateServer1.ini so that when queue managers are created using SimpleCreateQM, auto-registration is triggered. This section explains which keywords are required in the registry section of the `ini` files, and where to use the entity’s one-time-use certificate request PIN.

Prior to using message-level security to protect messages using MQeMTrustAttribute, the application must use private registry services to ensure that the initiating and recipient entities have credentials. This requires the following steps:

1. Setup and start an instance of MQe mini-certificate issuance service.
2. Add the name of the application entity, and allocate the entity a one-time-use certificate request PIN.
3. Use a program similar to the pseudo-code fragment below to trigger auto-registration of the application entity . This creates the entity’s credentials and saves them in its private registry.

```
/* SIMPLE MQePrivateRegistry FRAGMENT*/
  try
     {
     /* setup PrivateRegistry parameters  */
     String EntityName         = "Bruce";
     String EntityPIN            = "11111111";
     Object KeyRingPassword       = "It_is_a_secret";
     Object CertReqPIN            = "12345678";
     Object CAIPAddrPort          = "9.20.X.YYY:8082";
     /* instantiate and activate a
      Private Registry. */
     MQePrivateRegistry preg  = new MQePrivateRegistry( );
     preg.activate( EntityName,
    /* entity name               */
                    ".//MQeNode_PrivateRegistry",
    /* directory root  */
                    EntityPIN,
    /* private reg access PIN     */
                    KeyRingPassword,
    /* private credential keyseed */
                    CertReqPIN,
    /* on-time-use Cert Req PIN   */
```

```
                CAIPAddrPort );
  /* addr and port MiniCertSvr  */
   trace(">>> PrivateRegistry activated OK ...");
   }
catch (Exception e)
   {
   e.printStackTrace( );
   }
```

Private registry usage scenario:

The primary purpose of MQe's private registry is to provide a private repository for MQe authenticatable entity credentials. An authenticatable entity's credentials consist of the entity's mini-certificate (encapsulating the entity's public key), and the entity's keyring protected private key.

Typical usage scenarios need to be considered in relation to other MQe security features:

Queue-based security with MQeWTLSCertAuthenticator
: Whenever queue-based security is used, where a queue attribute is defined with MQeWTLSCertAuthenticator, mini-certificate based mutual authentication, the authenticatable entities involved are MQe owned. Any queue manager that is to be used to access messages in such a queue, any queue manager that owns such a queue and the queue itself are all authenticatable entities and need to have their own credentials. By using the correct configuration options and setting up and using an instance of MQe mini-certificate issuance service, auto-registration can be triggered when the queue managers and queues are created, creating new credentials and saving them in the entities' own private registries.

Message-level security with MQeMTrustAttribute
: Whenever message-level security is used with MQeMTrustAttribute, the initiator and recipient of the MQeMTrustAttribute protected message are application owned authenticatable entities that must have their own credentials. In this case, the application must use the services of MQePrivateRegistry (and an instance of MQe mini-certificate issuance service) to trigger auto-registration to create the entities' credentials and to save them in the entities' own private registries.

Private registry and authenticatable entity:

Queue-based security that uses mini-certificate based mutual authentication, and message-level security that uses digital signature, have triggered the concept of authenticatable entity. In the case of mutual authentication it is normal to think about the authentication between two users but, messaging generally has no concept of users. The normal users of messaging services are applications, and they handle the user concept.

MQe abstracts the concept of target of authentication from user to authenticatable entity. This does not exclude the possibility of authenticatable entities being people, but this would be application selected mapping.

Internally, MQe defines all queue managers that can either originate or be the target of mini-certificate dependent services as authenticatable entities. MQe also defines queues defined to use mini-certificate based authenticators as authenticatable entities. So queue managers that support these services can have

one authenticatable entity (the queue manager only), or a set of authenticatable entities (the queue manager and every queue that uses certificate based authenticator).

MQe provides configurable options to enable queue managers and queues to auto-register as an authenticatable entity. MQe private registry service, MQePrivateRegistry provides services that enable an MQe application to auto-register authenticatable entities and manage the resulting credentials.

All application-registered authenticatable entities can be used as the initiator or recipient of message-level services protected using MQeMTrustAttribute.

Authenticatable entity credentials:

To be useful every authenticatable entity needs its own credentials. This provides two challenges, firstly how to execute registration to get the credentials, and secondly where to manage the credentials in a secure manner. MQe private registry services help to solve these two problems. These services can be used to trigger auto-registration of an authenticatable entity creating its credentials in a secure manner and they can also be used to provide a secure repository.

Private registry (a descendent of base registry) adds to base registry many of the qualities of a secure or cryptographic token. For example, it can be a secure repository for public objects (mini-certificates) and private objects (private keys). It provides a mechanism to limit access to the private objects to the authorized user. It provides support for services (for example digital signature, RSA decryption) in such a way that the private objects never leave the private registry. Also, by providing a common interface, it hides the underlying device support.

Auto-registration:

MQe provides default services that support auto-registration. These services are automatically triggered when an authenticatable entity is configured; for example when a queue manager is started, or when a new queue is defined, or when an MQe application uses MQePrivateRegistry directly to create a new authenticatable entity.

When registration is triggered, new credentials are created and stored in the authenticatable entity's private registry. Auto-registration steps include generating a new RSA key pair, protecting and saving the private key in the private registry; and packaging the public key in a new-certificate request to the default mini-certificate server.

Assuming the mini-certificate server is configured and available, and the authenticatable entity has been pre-registered by the mini-certificate server (is authorized to have a certificate), the mini-certificate server returns the authenticatable entity's new mini-certificate, along with its own mini-certificate. These mini-certificates, together with the protected private key, are stored in the authenticatable entity's private registry as the entity's new credentials.

While auto-registration provides a simple mechanism to establish an authenticatable entity's credentials, in order to support message-level protection, the entity requires access to its own credentials (facilitating digital signature) and to the intended recipient's public key (mini-certificate).

Public registry service

This section describes the public registry service provided by MQe.

MQe provides default services facilitating the sharing of authenticatable entity *public credentials* (mini-certificates) between MQe nodes. Access to these mini-certificates is a prerequisite for message-level security. MQe public registry, also a descendent of base registry, provides a publicly accessible repository for mini-certificates. This is analogous to the personal telephone directory service on a mobile phone, the difference being that it is a set of mini-certificates of the authenticatable entities instead of phone numbers.

MQe public registry is not a purely passive service. If accessed to provide a mini-certificate that is does not hold, and if the public registry is configured with a valid home server, the public registry automatically attempts to get the requested mini-certificate from the public registry of the home server. It also provides a mechanism to share a mini-certificate with the public registry of other MQe nodes. Together these services provide the building blocks for an intelligent automated mini-certificate replication service that can facilitates the availability of the right mini-certificate at the right time.

Public registry usage scenario:

A typical scenario for the use of the public registry would be to use these services so that the public registry of a particular MQe node builds up a store of the most frequently needed mini-certificates as they are used.

A simple example of this is to setup an MQe client to automatically get the mini-certificates of other authenticatable entities that it needs, from its MQe home server, and then save them in its public registry.

Secure feature choices:

It is the Solution creator's choice whether to use the public registry active features for sharing and getting mini-certificates between the public registries of different MQe nodes.

The alternative to this intelligent replication may be to have an out-of-band utility to initialize an MQe node's public registry with all required mini-certificates before enabling any secure services that uses them.

Selection criteria:

Out-of-band initialization of the set of mini-certificates available in an MQe node's public registry may have advantages over using the public registry active features in the case where the solution is predominantly asynchronous and the synchronous connection to the MQe node's home server may be difficult. But in the case where this connection is more likely to be available, the public registry's active mini-certificate replication services are useful tools to automatically maintain the most useful set of mini-certificates on any MQe node public registry.

Example - public registry:

```
/*SIMPLE MQePublicRegistry shareCertificate FRAGMENT */
   try {
   String EntityName = "Bruce";
   String EntityPIN = "12345678";
   Object KeyRingPassword = "It_is_a_secret";
```

```
   Object CertReqPIN = "12345678";
   Object CAIPAddrPort = "9.20.X.YYY:8082";
/*instantiate and activate PublicReg */
   MQePublicRegistry pubreg = new MQePublicRegistry();
   pubreg.activate("MQeNode_PublicRegistry",".\\");
/* auto-register Bruce1,Bruce2...Bruce8 */
/* ... note that the mini-certificate issuance service must */
/* have been configured to allow the auto-registration    */
   for (int i = 1; i < 9; i++)
   {
   EntityName = "Bruce"+(new Integer(i)).toString();
   MQePrivateRegistry preg = new MQePrivateRegistry();
/* activate() will initiate auto-registration */
   preg.activate(EntityName, ".\\MQeNode_PrivateRegistry",
     EntityPIN, KeyRingPassword, CertReqPIN, CAIPAddrPort);
/* save MiniCert from PrivReg in PubReg*/
    pubreg.putCertificate(EntityName,
       preg.getCertificate(EntityName ));
/*before share of MiniCert */
   pubreg.shareCertificate(EntityName,
     preg.getCertificate(EntityName ),"9.20.X.YYY:8082");
   preg.close();
   }
   pubreg.close();
   }
   catch (Exception e)
   {
   e.printStackTrace();
   }
```

Note:

1. It is not possible to activate a registry instance more than once, hence the example above demonstrates the recommended practice of accessing a private registry by creating a new instance of MQePrivateRegistry, activating the instance, performing the required operations and closing the instance.
2. If you want to share certificates using a public registry on the home-server, the public registry must be called `MQeNode_PublicRegistry`.

Mini-certificate issuance service

The ES03 MQe SupportPac, ″MQe WTLS Mini-Certificate Server″ is available as a separate free download from *http://www.ibm.com/software/ts/mqseries/txppacs/*. MQe includes a default *mini-certificate issuance service* that can be configured to satisfy private registry auto-registration requests. With the tools provided, a solution can setup and manage a mini-certificate issuance service so that it issues mini-certificates to a carefully controlled set of entity names. These are a prerequisite for MQeMTrustAttribute-based message-level security. The characteristics of this issuance service are:

- Management of the set of registered authenticatable entities.
- Issuance of mini-certificates. The mini-certificate conforms to the WAP WTLS specification.
- Management of the mini-certificate repository.

The tools provided in the ES03 SupportPac enable a mini-certificate issuance service administrator to authorize mini-certificate issuance to an entity by registering its entity name and registered address and defining a one-time-use *certificate request PIN*. This would normally be done after off line checking to validate the authenticity of the requestor. The certificate request PIN can be posted to the intended user, as bank card PINs are posted when a new card is issued. The

user of the private registry (for example the MQe application or MQe queue manager) can then be configured to provide this certificate request PIN at startup time. When the private registry triggers auto-registration, the mini-certificate issuance service validates the resulting new certificate request , issues the new mini-certificate and then resets the registered certificate request PIN so it cannot be reused. All auto-registration of new mini-certificate requests is processed on a secure channel.

We recommend that you refer to the MQe_MiniCertificateServer documentation included in the ES03 SupportPac, ″MQe WTLS Mini-Certificate Server″, for more details of how to install and use the WTLS digital certificate issuance service for MQe.

Renewing mini-certificates:

The certificates issued for an entity by the mini-certificate issuance service are valid for one year from the date of issue and it is advisable to renew them before they expire. Renewed certificates are obtained from the same mini-certificate issuance service. Before requesting a renewal, the request must be authorized with the issuance service and a one-time-use certificate request PIN obtained, in just the same way as for the initial certificate issuance. When you use the server to obtain the PIN for renewal, remember that you are updating the entity, not adding it.

When a certificate is issued for an entity, a copy of the mini-certificate server's own certificate is issued with it. This is needed to check the validity of other certificates. With versions of MQe earlier than 1.2, the certificate server's certificate could expire before the entity's certificate. If this happens you can renew the server's certificate by requesting a renewal of the entity's certificate; a new copy of the mini-certificate server's certificate will be returned along with the entity's certificate. From mini-certificate server Version 1.2, the mini-certificate server's certificate will expire later than the entity's certificate.

The class com.ibm.mqe.registry.MQePrivateRegistryConfigure contains a method **renewCertificates()** which can be used to request renewed certificates. This is used in the example program examples.certificates.RenewWTLSCertificates, which implements a command-line program that requests renewed certificates from the issuance service

The program has four compulsory parameters:

```
RenewWTLSCertificates <entity> <ini file> <MCS addr> <MCS Pin>
```

where:

entity is the name of the entity for which a renewed certificate is required. This should be either a queue manager, a queue or other authenticatable entity. The name of a queue should be specified as `<queue manager>+<queue>`, for example `myQM+myQueue`.

ini file
is the name of a configuration file that contains a section for the registry. This is typically the same configuration file that is used for the queue manager. For a queue, this typically the configuration file for the queue manager that owns the queue.

MCS addr
is the host name and port address of the mini-certificate server (for example: `myServer:8085`)

MCS Pin
: is the one-time use PIN issued by the mini-certificate server administrator to authorize this renewal request.

Obtaining new credentials (private and public keys):

When you renew a certificate, you get an updated certificate for your existing public key. This allows you to continue to use your existing private and public key pair. If you want to change your private and public key pair, you must request new credentials. This includes a request to the mini-certificate issuance service for a new public certificate embodying the new public key. Before requesting a certificate for the new credentials, the request must be authorized with the issuance service and a one-time-use certificate request PIN must be obtained, in the same way as for the initial certificate issuance. (When you use the server to obtain the PIN for the new certificate, remember that you are updating the entity, not adding it.)

The class com.ibm.mqe.registry.MQePrivateRegistryConfigure contains a method **getCredentials()** which can be used to request new credentials. This is used in the example program examples.install.GetCredentials, which implements a GUI program that requests new credentials from the issuance service.

Note: When new credentials are issued, the existing ones are archived in the registry. You will no longer be able to decrypt messages created using your earlier credentials. The new certificate will not validate a digital signature (used with MQeMTrustAttribute) created with your earlier credentials.

Listing mini-certificates:

It can be useful to list the certificates in a registry, for example to check on their expiry dates. You can do this using methods in the class com.ibm.mqe.attributes.MQeListCertificates. These are used in the example program examples.certificates.ListWTLSCertificates, which implements a command-line program that lists certificates.

The program has one compulsory and three optional parameters:

```
ListWTLSCertificates <reg Name>[<ini file>] [<level>] [<cert names>]
```

where:

regName
: is the name of the registry whose certificates are to be listed. It can be a private registry belonging to a queue manager, a queue or another entity; it can be a public registry, or (for the administrator) it can be the mini-certificate server's registry. If you want to list the certificates in a queue's registry, you must specify its name as `<queue manager>+<queue>`, for example `myQM+myQueue`. If you want to list the certificates in a public registry, it must have the name `MQeNode_PublicRegistry`, it will not work for a public registry with any other name. The name of the mini-certificate server's registry is `MiniCertificateServer`.

ini file
: is the name of a configuration file that contains a section for the registry. This is typically the same configuration file that is used for the queue manager or mini-certificate server. For a queue, this is typically the

configuration file for the queue manager that owns the queue. This parameter should be specified for all registries except public registries, for which it can be omitted.

level is the level of detail for the listing. This can be:

-b or -brief
prints the names of the certificate, one name per line

-n or -normal
prints the names of the certificates, one per line, followed by their type (old or new format)

-f or -full
prints the names of the certificates, their type, and some of the contents

This parameter is optional and if omitted the "normal" level of detail is used.

cert names
is a list of names of the certificates to be listed. It starts with the flag `-cn` followed by names of the certificates, for example: `-cn ExampleQM putQM`. If this parameter is used, only the named certificates are listed. If this parameter is omitted, all the certificates in the registry are listed.

Performance

MQe can be used in a number of different configurations, and the performance you can expect will vary a great deal depending on your adapters and manner of use.

The main thing to be aware of when configuring MQe is that disk accesses are the single biggest cause of slowdown in an MQe system. All unnecessary disk accesses should be designed out from the beginning.

Try to split the messages that you'll be dealing with into messages that it's important are persistent and messages that do not need to be persistent. The persistent messages need to use a disk fields adapter for storage, but the non-persistent ones should use a memory fields adapter. Non-persistent messages stored in memory can go around 100 times faster than messages stored to disk.

When possible, distribute queues across different physical hard discs, so that reads and writes to different queues can take place using different hardware and happen simultaneously.

When multiple clients are accessing a single server, use multiple queues, as only one client can use a queue at a time. Avoid very large numbers of queues, as this increases the time to do any MQe access.

Keep polling systems such as trigger transmit rules or home server queue polls to a minimum. Unless you need a specific performance characteristic, the intervals between these can often be configured to be quite large. If you are using them together, then the trigger transmit rule, which is only used to automatically recover a home server queue from network stoppage can often be set to have a much larger interval. If you are designing an application that makes use of home server queues and you are using a trigger transmission rule, then consider replacing it with a user interaction to cause the trigger transmission.

Most JVMs can have their initial memory settings tweaked. These settings are often on -msX and -mxX. Executing java -X will give you more information. Try increasing the initial and maximum heap size to as much as you can without causing the machine to start paging.

If you are running some application with a queue manager that is under a lot of external load, be aware that your own application may suffer from reduced performance as many threads to deal with incoming messages are started. Making sure your own application is multithreaded can reduce this problem.

Errors and error handling

Overview of errors and error handling in Java and C

This chapter describes what happens if an error occurs within the Java and C codebases.

Error handling in Java

Errors within the Java codebase are handled using exceptions. The MQe Java API Programming Reference documents all of the exception codes that the MQe Java code can return in the following classes:

- com.ibm.mqe.MQeExceptionCodes
- com.ibm.mqe.mqbridge.MQeBridge.ExceptionCodes

Error handling in C

The C codebase indicates errors using *Return* and *Reason* codes. The C code does not have any exception handling mechanism, as in C++. MQe does not use the operating system error handling functions. An `MQeExceptBlock` handles errors and returns values from the functions. An application is free to install any operating system exception handlers that it requires.

The specific nature of an error condition is returned using two values, `MQERETURN` and `MQEREASON`. `MQERETURN` determines the general area in which the application failed, and distinguishes between warnings and errors. You can ignore warnings, but you must not ignore errors. With errors, your application needs to solve the problem in order to continue safely.

`MQERETURN` and `MQEREASON` are both returned in the `MQeExceptBlock`. The `MQERETURN` value is also the return value from the function.

Code structure

The MQe_nativeReturnCodes.h header file lists all of the return and reason codes. They are divided into function area and then by error or warning. For example, `MQERETURN_QUEUE_MANAGER_ERROR` and `MQERETURN_QUEUE_MANAGER_WARNING`. Warnings indicate that a situation can be ignored.

Exception block

The `MQeExceptBlock` structure is used to pass the return code and reason code, generated by a function call, back to the user. If a function call does not return `MQERETURN_OK`, use the ERC macro to get the reason code.

MQe ships two macros:

EC This macro resolves to the return code in the exception block structure.

ERC This macro resolves to the reason code in the exception block structure.

The convention within MQe is that a pointer to an exception block is passed first on a new function. A pointer to the object handle is passed second, followed by any additional parameters. On subsequent calls, the object handle is the first parameter passed, and the pointer to the exception block is second, followed by any additional parameters.

The structure of the exception block, as shown in the following example, is MQeExceptBlock_st.

```
struct MQeExceptBlock_st
  {
      MQERETURN    ec;
            /* return code*/
      MQEREASON    erc;
            /* reason code*/
      MQEVOID*      reserved;
            /* reserved for internal use only*/
  }
```

It is recommended that you allocate the Exception Block on the stack, rather than the heap. This simplifies possible memory allocations, although there are no restrictions on allocating space on the heap. The following code demonstates how to do this:

```
MQERETURN rc
MQeExceptBlock exceptBlock;
/*.....initialisation*/
rc = mqeFunction_anyFunction(&exceptBlock,
/*parameters go here*/);
if (MQERETURN_OK ! = rc) {
printf("An error has occured, return code =
      %d, reason code =%d \n",
      exceptBlock.ec exceptBlock.erc);
}else {
}
```

All API calls need to take exception blocks. The C Bindings codebase permits NULL to be passed to an API call. However, this feature is deprecated in the C codebase and, therefore, not recommended.

You should use a different exception block for each thread in the application.

Note: If an error is not corrected, subsequent API calls can put the system in an unpredictable state.

Useful macros

A number of macros help to access the exception block:

SET_EXCEPT_BLOCK

Sets the return and reason codes to specific values, for exampe:

```
MQeExceptBlock exceptBlock;
SET_EXCEPT_BLOCK(&exceptBlock,
        MQERETURN_OK,
        MQEREASON_NA);
```

SET_EXCEPT_BLOCK_TO_DEFAULT
Sets return and reason codes to non-error values, for example:

```
MQeExceptBlock exceptBlock;
SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);
```

EC Accesses the return code, for example:

```
MQeExceptBlock exceptBlk;
/*MQe API call */
MQERETURN returncode;
returnCode = EC(&exceptBlock);
```

ERC Accesses the reason code, for example:

```
MQeExceptBlock exceptBlk;
/*MQe API call*/
MQEREASON reasoncode;
MQEREASON reasonCode = ERC(&exceptBlock);
```

NEW_EXCEPT_BLOCK
Can create a temporary exception block. This is useful for temporary clean-up operations.

Java programming samples

Introduction to the set of Java examples provided with MQe

This topic provides a brief description of the set of Java programming examples provided with MQe. Each example demonstrates how to use or extend a feature of MQe, and most of them are described more fully in the relevant topics in this information center.

Adapters (examples.adapters)

This package provides two example classes that conform to the MQe adapters specification.

MQeDiskFieldsAdapter
This example class is identical in functionality to the disk fields adapter found in com.ibm.mqe.adapters. It supports the reading and writing of data on the local file store.

WESAuthenticationGUIAdapter
Wrappers the WESAuthenticationAdapter found inside com.ibm.mqe.adapters. This example enhances the WESAuthenticationAdapter by displaying a dialog box that prompts the user for login information when connecting to a WebSphere Everyplace proxy.

Command line administration (examples.administration.commandline)

This package contains a suite of example tools for creating base MQe objects from the command line. Each program is a simple example of how to send administration messages and how to interpret the replies.

Using these tools and a script, you can reliably set up exactly the same configuration on a number of machines.

GUI administration (examples.administration.console)

This package contains a set of classes that implement a simple graphical user interface (GUI) for managing MQe resources.

Admin
Front end to the example administration GUI.

Additionally there is a suite of classes that provides the graphical user interface for each MQe managed resource.

The GUI can be invoked in any of the following ways:
- Using the batch file ExamplesAdminConsole.bat
- From the command line:
```
java examples.administration.console.Admin
```
- From a button on the example server `examples.awt.AwtMQeServer`

Simple administration (examples.administration.simple)

This package contains a set of examples that show how to use some of the administrative features of MQe in your programs. As with the application examples, these examples can work with either a local or a remote queue manager.

Example1
Create and delete a queue.

Example2
Add a connection definition for a remote queue manager.

Example3
Inquire on the characteristics of a queue manager and the queues it owns.

ExampleAdminBase
The base class that all administration examples inherit from.

Interaction with a queue manager (examples.application)

This package contains a set of examples that demonstrate various ways to interact with a queue manager. These include putting a message to and getting a message from a queue. All the examples can be used with either a local queue manager or a remote queue manager. Before you can use any of these applications, the queue managers that are to be used must be created. You can use the CreateExampleQM.bat batch file on Windows, or the CreateExampleQM shell script on UNIX, to create queue managers `ExampleQM` (see Verifying your installation).

Example1
Simple put and get of a message.

Example2
Put several messages and then get the second one using a match field.

Example3
Use a message listener to detect when new messages arrive.

Example5
Lock messages then get, unlock, and delete them.

Example6
Simple put and get of a message using assured message delivery.

Example7
Simple put and get of a message through a Websphere Everyplace proxy.

ExampleBase
The base class that all application examples inherit from.

These examples can be run as follows:

Windows
Using batch file ExamplesMQeClientTest.bat

```
ExamplesMQeClientTest <JDK> <example no>
    <remoteQMgrName> <localQMgr ini file>
```

UNIX Using shell script ExamplesMQeClientTest

```
ExamplesMQeClientTest  <example no>
<remoteQMgrName> <localQMgr ini file>
```

where

<JDK> is the name of the Java environment. The default is IBM

Note: This parameter is not used on UNIX.

<example no>
is the number of the example to run (suffix of the name of the example). The default is 1 (Example1).

<remoteQMgrName>
is the name of the queue manager that the application should work with. This can be the name of the local or a remote queue manager. If it is a remote queue manager, a connection must be configured that defines how the local queue manager can communicate with the remote queue manager.

By default the local queue manager is used, as defined in ExamplesMQeClient.ini.

<localQMgrIniFile>
is an ini file containing startup parameters for a local queue manager. By default ExamplesMQeClient.ini is used.

For more details on how to write applications that interact with a queue manager see “Queue manager operations” on page 49.

Security (examples.attributes)

This package contains a set of classes that show how to write additional components to extend MQe security. However, they are not designed to be used for asynchronous messaging and do not provide very strong security.

NTAuthenticator
An authenticator that authenticates a user to the Windows NT security database. To authenticate correctly the user must have the following User Rights set on the target NT system:

- Act as part of the operating system
- Logon locally
- Logon as a service

The NT authenticator uses the Java native interface (JNI) to interact with Windows NT security. The code for this can be found in the examples.nativecode directory. The dll built from this code must be placed in the PATH of the NT machine that owns the target resource.

UnixAuthenticator
An authenticator that authenticates a user using the UNIX password or shadow password system. The UNIX authenticator uses the JNI to interact with the host system. The code for this can be found in the examples.nativecode directory. If your system supports the shadow password file, you must recompile this native code with the `USE_SHADOW` preprocessor flag defined. You must also ensure the code has sufficient privileges to read the shadow password file when it executes. This example does not work if your system uses a distributed logon service (such as Lightweight Directory Access Protocol (LDAP)).

LogonAuthenticator
Base logon authentication support.

UseridAuthenticator

Support for base *userID* authentication.

This example requires a UserIDS.txt file as input. This file must have the format:

```
[UserIDs]

User1Name=User1Password

...

UserNName=UserNPassword
```

See "Security" on page 198 for more detailed information about the MQe security features.

Adding a small GUI to an application (examples.awt)

This package provides a toolkit for building applications that require a small graphical interface. It also contains example applications that provide a graphical front end to MQe functions.

AwtMQeServer
A graphical front end to the `examples.queuemanager.MQeServer` example. The MQeTraceResourceGUI class provides a resource bundle that contains internationalized strings for use by the GUI. MQeTraceResourceGUI is in package examples.trace.

You can use the batch file ExamplesAwtMQeServer.bat to run this application.

See "Server queue managers" on page 63 for more details about running a queue manager in a server environment.

AwtMQeTrace
A graphical front end to examples.trace.MQeTrace.

See "Java Message Service (JMS)" on page 177 for more information about the MQe trace facility.

Classes **AwtDialog**, **AwtEvent**, **AwtFormat**, **AwtFrame**, and **AwtOutputStream** provide a toolkit for building small footprint awt-based graphical applications. These classes are used by many of the graphical MQe examples.

Managing mini-certificates (examples.certificates)

This package contains examples for managing mini-certificates. See “Mini-certificate issuance service” on page 241 for more information on these examples, and using mini-certificates.

ListWTLSCertificates
: This example uses methods in the class com.ibm.mqe.attributes.MQeListCertificates to implement a command line program which lists mini-certificates in a registry, to varying levels of detail.

RenewWTLSCertificates
: This example uses methods in the class com.ibm.mqe.registry.MQePrivateRegistryConfigure to implement a command line program which renews mini-certificates in a registry. This should be used only on a private registry.

Logging events (examples.eventlog)

This package contains some examples that demonstrate how to log events to different facilities.

LogToDiskFile
: Write events to a disk file.

LogToNTEventLog
: Write events to the Windows NT event log. This class uses the JNI to interact with the Windows NT event log. The code for this is in the examples.nativecode directory.

LogToUnixEventLog
: Write events to the UNIX event log (which is normally /var/adm/messages). This class uses the JNI to interact with the UNIX event logging system. The code for this can be found in the examples.nativecode directory. The syslog daemon on your system should be configured to report the appropriate events.

Creating and deleting queue managers (examples.install)

This package contains a set of classes for creating and deleting queue managers.

DefineQueueManager
: A GUI that allows the user to select options when creating a queue manager. When the options have been selected, this example creates an ini file containing the queue manager startup parameters, and then creates the queue manager.

CreateQueueManager
: A GUI program that requests the name and directory of an ini file that contains queue manager startup parameters. When the name and directory are provided, a queue manager is created.

SimpleCreateQM
: A command line program that takes a parameter that is the name of an ini

file that contains queue manager startup parameters. It also optionally takes a parameter that is the root directory where queues are stored. Provided a valid ini file is found, a queue manager is created.

DeleteQueueManager
A GUI program that takes the name of an ini file that contains queue manager startup parameters. Provided a valid ini file is found, the queue manager is deleted.

SimpledDeleteQM
A command line program that takes a parameter that is the name of an ini file that contains queue manager startup parameters. Provided a valid ini file is found, the queue manager is deleted.

GetCredentials
A GUI program that takes the name of an ini file that contains queue manager startup parameters. Provided a valid ini file is found, new credentials (private/public key pair and public certificate) are obtained for the queue manager. The mini-certificate server must be running and the request for a new certificate must have been authorized for this to succeed (see “Mini-certificate issuance service” on page 241).

All the configuration files use the resources and utilities provided in **ConfigResource**, and **ConfigUtils**.

For more details about creating and deleting queue managers, see “Queue manager operations” on page 49.

Extending the MQ bridge (examples.mqbridge.awt)

This package contains a set of classes that show how to use and extend the MQ bridge. Some of the examples extend other MQe examples.

AwtMQBridgeServer
This is an example of a graphical interface for the underlying examples.mqbridge.queuemanager.MQBridgeServer class.

The MQBridgeServer class source code demonstrates how to add bridge functionality to your MQe server program, following these guidelines.

To start the bridge enabled server:
1. Instantiate the base MQe queue manager, and start it running.
2. Instantiate a com.ibm.mqe.mqbridge.MQeMQBridges object, and use its activate() method, passing the same .ini file information as you passed to the base MQe queue manager.

The bridge function is then usable.

To stop the bridge-enabled server:
1. Disable the bridge function by calling the MQeMQBridges.close() method. This stops all the current MQ bridge operations cleanly, and shuts down all the MQ bridge function.
2. Remove your reference to the MQeMQBridges object, allowing it to be garbage-collected.
3. Stop and close the base MQe queue manager.

ExamplesAwtMQBridgeServer.bat
: This file provides an example of how to invoke the MQBridgeServer using the Awt server. It also shows how to control the initial settings of the AwtMQBridgeTrace module.

ExamplesAwtMQBridgeServer.ini
: This file provides an example configuration file for a queue manager that supports MQ bridge functionality.

Administering objects for an MQ bridge (examples.mqbridge.administration.commandline)

This package contains a suite of example tools, similar to those in the examples.administration.commandline package, designed to administer the objects required for an MQ bridge.

Testing communication between MQ and MQe (examples.mqbridge.application.GetFromMQ)

The example programs in this package are useful for proving that MQe and MQ can communicate with each other. These examples are MQ bindings programs that use the Java classes and are driven by a simple command-line syntax.

GetFromMQ
: This class destructively reads any message appearing on a specified MQ queue, and provides timing statistics on when the message arrives. Optionally the message content can be dumped to the standard output screen.

  This example is useful when testing a link between MQe and MQ, to see what throughput is being achieved between the two systems. Scripts dealing with connectivity between MQe and MQ can refer to and use this class.

PutFromMQ
: This class puts a message to an MQ queue, such that the user can specify the target queue and the target queue manager. It specifically uses the long form of the MQQueueManager.accessQueue() method to make use of any MQe queue manager alias definitions that might be defined on the MQ queue.

MQe interface (examples.mqeexampleapp)

This package contains two example applications to aid your understanding of the MQe interface. The example code can be split into 3 parts:

The message service (examples.mqeexampleapp.messageservice)
: This runs MQe, controls a queue manager and performs functions such as queue creation and message sending. This is the core of the examples and allows them to be written with minimal calls to the MQe API. This also means that to see the code required to create a local queue for example, a user can simply look at the relevant function within MQeMessageService.

Example 1: The message pump (examples.mqeexampleapp.msgpump)
: This is a very simple application consisting of a single server and client. The client is set to send a message to the server every 3 seconds which, when received by the server, will be displayed to the user. Queues are

asynchronous. Implementations of the client are available for both MIDP and J2SE, while the server is only available for J2SE.

Example 2: The text application (examples.mqeexampleapp.textapp)
This is slightly more complex than the first example, consisting of 2 servers and a client. When initiating, the client is required to register with the registration server. The registration server adds the client to a store-and-forward queue on the gateway server and replies with a success or failure message. The client can then send user-defined messages to the gateway server (which it will display). The aim of this application is to show how a separate server can be used to create resources necessary for a new client on the system to aid scalability of large MQe networks.

JNI implementation (examples.nativecode)

Several of the examples require access to operating system facilities on Windows NT, or UNIX (AIX and Solaris). MQe accesses these functions using the JNI. For Windows, the code in the examples\native directory provides the JNI implementation required by `examples.attributes.NTAuthenticator` and `examples.eventlog.LogToNTEventLog`. For UNIX, the code in the file examples/native/JavaUnix.c provides the JNI implementation required by the `examples.attributes.UnixAuthenticator` and `examples.eventlog.LogToUnixEventLog`.

Running a QM as a client, server, or servlet (examples.queuemanager)

A queue manager can run in many different types of environment. This package contains a set of examples that allow a queue manager to run as a client, server, or servlet:

MessageWaiter
An example of how to wait for messages without using the deprecated waitFormessage method.

MQeClient
A simple client typically used on a device.

MQePrivateClient
A client that can be used with secure queues and secure messaging.

MQeServer
A server that can connect concurrently to multiple queue managers (clients or servers). This is typically used on a server platform. Batch file ExamplesAwtMQeServer.bat can be used to run the `examples.awt.AwtMQeServer` example which provides a graphical front end to this server.

MQePrivateServer
Similar to MQeServer but allows the use of secure queues and secure messaging.

MQeServlet
An example that shows how to run a queue manager in a servlet.

MQeChannelTimer
An example that polls the channel manager so that it can time-out idle channels.

MQeQueueManagerUtils
A set of helper methods that configure start various MQe components.

For more details about running queue managers in different environments see “Starting queue managers” on page 57. For details on queue managers that provide an environment for secure queues and messaging (MQePrivateClient and MQePrivateServer), see “Security” on page 198.

Rules classes (examples.rules)

You can control and extend the base MQe functionality using rules. Some components of MQe invoke rules classes. These rules provide a means of changing the functionality of the component. This package contains the following example rules classes:

ExamplesQueueManagerRules
Example queue manager rules class makes regular attempts to transmit any held messages. See “Message delivery” on page 82 for more details.

AttributeRule
Example attribute rule that controls the use of attributes.

Trace handling (examples.trace)

This package contains an example trace handler that can be used for debugging an application during development, and for tracing a completed application.

MQeTrace
The base MQe trace class.

AwtMQeTrace, which is in the examples.awt package, provides a graphical front end to the MQeTrace class.

MQeTraceResource
A resource bundle that contains trace messages that can be output by MQe.

MQeTraceResourceGUI
This class contains all the translatable text for the trace window controls.

Deploying your application

Packaging and deployment

MQe is a flexible messaging system that can be deployed to a wide variety of operating systems and devices.

This section provides information to assist in the build, packaging and deployment of MQe.

It is split into two sections covering the Java code base and the native code base.

Because MQe can be deployed on a variety of devices, operating systems, and runtimes, it is not possible to detail each application. Hence in some topics only a brief outline and introduction is provided.

Java deployment

The MQe Java code base can be deployed onto a large variety of Java runtimes. These include:

- J2ME CLDC/MIDP
- J2ME CDC/Foundation
- PersonalJava V1.1
- Java 1.1
- J2SE 1.2 (or later)
- IBM® WebSphere Studio Custom Environment (WSCE) jclGateway (or better)

The way that MQe, the application and other classes are packaged and deployed is dependant on the type of Java runtime, the operating system and processor type of the device that is being deployed to.

The following topics provide information to assist in packaging and deploying Java based MQe applications to different environments.

Supplied jar files

When deploying MQe applications, you are recommended to pack the minimum set of classes required by the application into compressed jar files. This ensures that the application requires the minimum system resources. MQe provides the following examples of how the MQe classes can be packaged into .jar files. These examples are in the<MQeInstallDir>\Java\Jars directory of a standard MQe installation.

There are three types of jar file; *base*, *extension*, and *other*:

- The base jar files allow a usable queue manager to be created, administered and run
- The extension jar files can be used in addition to the base jar files to provide additional capability
- The other jar files include example, and core, sets of classes for you to use as a base for your development

© Copyright IBM Corp. 2000, 2004

Base jar files

MQeBase.jar
: Contains classes that provide for a basic queue manager running in client and server mode on a J2ME CDC/Foundation or J2SE or better Java runtime.

MQeMidp.jar
: Similar to MQeBase.jar but for use on a J2ME CLDC/MIDP Java runtime. Allows a queue manager to run in client mode. All MIDP compliant classes are included in this jar. No extension jars can be used with this one, as they are not MIDP compliant.

MQeGateway.jar
: Contains classes that provide for a basic queue manager running in client, server and bridge mode on a J2SE or better Java runtime.

Extension jar files

MQeJMS.jar
: Contains the classes that extend an MQe queue manager to provide a JMS programming interface.

MQeRetail.jar
: Contains extra classes for use in retail environments. In particular, these classes are useful on a 4690 retail system.

MQeSecurity.jar
: A set of classes that are used to provide both queue and message based security. It contains a set of cryptors, compressors and authenticators.

MQeBindings.jar
: This file contains all C bindings specific information. It is required if access to a Java queue manager from a C application is needed (only on Win32 platforms).

MQeMigration.jar
: Contains classes that assist in migrating from an earlier version of MQe.

MQeDeprecated.jar
: This contains all of the deprecated class files that are no longer needed by an MQe application. These deprecated class files help you run applications written using a previous version of MQe, without making any changes.

MQeDiagnostics.jar
: This file helps to diagnose problems with MQe classes. It contains tooling to search the class path to find out the level of each class found.

Other jar files

MQeExamples.jar
: A packaging of all the MQe examples into one jar file. This includes all of the examples supplied with MQe, but excludes the deprecated classes.

MQeCore.jar
: This contains a minimal set of classes. On its own it is not usable but it can be used as a base for building a small footprint MQe system. More details on reducing footprint can be found in the ″Optimizing footprint″ section.

Optimizing footprint

In many cases the supplied jar files can be used without change, however there are instances where this is not the case. In particular, on some environments where

footprint is limited, the set of classes that are deployed must be reduced to the smallest possible size. The supplied jar files are general purpose and contain more than is necessary for an optimized environment.

The table below separates the classes into groups associated with a particular function or configuration and will help determine which classes will be required to optimize an applications footprint. Using this table the minimum required set of classes can be deduced by taking the mandatory classes for the required categories and then adding in required optional classes for that category.

Due to the wide ranging set of Java runtimes that are now available, not all classes can run on all runtimes. The table lists all classes, and unless otherwise stated, each class will run on a J2SE runtime. Because of the differences between a J2SE and a J2ME runtime, some of the classes are not appropriate for a J2ME runtime. There are two columns marked with an X to show a class that can be used on J2ME MIDP or J2ME CDC/Foundation runtimes.

Table 12. Class optimization

Category	Detail		
Type	Details	Midp	CDC
Classes required (com.ibm.mqe)			
Mandatory classes			
	For all queue managers	X	X
MQe MQeAdapter MQeAttribute MQeAttributeDefaultRule MQeAttributeRule MQeAuthenticator MQeCompressor MQeCryptor MQeEnumeration MQeException MQeExceptionCodes MQeField MQeFields MQeKey MQeLoaderMQeProperties MQePropertyProvider MQeQueueControlBlock MQeQueueProxy MQeQueueManager MQeQueueManagerRule MQeResourceControlBlock MQeRule MQeRunnable MQeRunnableInstance MQeThread MQeThreadPool$1 MQeThreadPool$PooledThread MQeThreadPool$Target MQeThreadPool MQeTrace MQeTraceHandler MQeTraceInterface registry.MQeRegistry			
Registry type	One option in this category must be selected		

Table 12. Class optimization (continued)

	File registry	Add required: Storage adapter	X	X
		registry.MQeFileSession registry.MQeRegistrySession		
	Private registry w/o credentials	Add: File registry		X
		registry.MQePrivateRegistry registry.MQePrivateSession		
	Private registry with credentials	Add: Private registry w/o credentials		X
		attributes.MQeMiniCertRequest attributes.MQeSharedKey attributes.MQeWTLSCertificate		
		Mini-certificate management functions		X
		attributes.MQeListCertificates registry.MQePrivateRegistryConfigure		
	Public registry	Applicable to types of message-level security, Add: Private registry with credentials		X
		registry.MQePublicRegistry		
Queue manager type		For all types add required: Administration Storage adapters Message store Authenticators Cryptors Compressors Rules Security		
	Standalone qMgr.	No additional classes		
	Client qMgr.	Add required: Communications	X	X
		MQeTransporter adapters.MQeCommunicationsAdapter communications.MQeChannel communications.MQeChannelCommandInterface communications.MQeChannelControlBlock communications.MQeCommunicationsException communications.MQeCommunicationsManager communications.MQeConnectionDefinition communications.MQeListener communications.MQeListenerSlave		
	Server qMgr.	Add: Client qMgr. Add required: Communications		X
		Note: whilst MQeListener is not used in the Client, they need to be included when preverifying a J2ME application		
	Gateway qMgr.	Add: Server qMgr. Add required Communications Transformers		
		MQeBridgeLoadable MQeBridgeManager mqbridge.*		
Communications				

Table 12. Class optimization (continued)

TCP/IP w/o history & persistence			X
`adapters.MQeTcpipAdapter` `adapters.MQeTcpipLengthAdapter`			
TCP/IP with history & persistence	Add: TCP/IP w/o history and persistence		X
`adapters.MQeTcpipHistoryAdapter` `adapters.MQeTcpipHistoryAdapterElement`			
HTTP 1.0 Not to WES Proxy Authentication server			X
`adapters.MQeTcpipAdapter` `adapters.MQeTcpipHttpAdapter`			
HTTP To WES Proxy Authentication server			X
`adapters.MQeTcpipAdapter` `adapters.MQeWESAuthenticationAdapter`			
HTTP 1.1/1.0 J2ME	MIDP only	X	
`adapters.MQeMidpHttpAdapter`			
UDP			X

`adapters.MQeUdpipBasicAdapter$Initiator`
`adapters.MQeUdpipBasicAdapter$InternalAdapter`
`adapters.MQeUdpipBasicAdapter$Responder`
`adapters.MQeUdpipBasicAdapter$Writer`
`adapters.MQeUdpipBasicAdapter`			

Queue Types	For all queue types add required: Authenticators Cryptors Compressors Rules		
Local	Add: Storage adapter Message storage	X	X

`MQeAbstractQueueImplementation`
`MQeEventTrigger`
`MQeMessageEvent`
`MQeMessageListenerInterface`
`MQeQueue`
`MQeQueueRule (or replacement)`			

Remote	Add: Local queue (storage adapter & msg. storage only if needed)	X	X
`MQeRemoteQueue`			
Home server	Add: Remote queue (no storage adapter or msg. storage)	X	X
`MQeHomeServerQueue`			
Sore and forward	Add: Remote queue	X	X
`MQeStoreAndForwardQueue`			
Bridge queue	Add: Remote queue		

`mqbridge.MQeMQBridgeAdminMsg`
`mqbridge.MQeBridgeServices`
`mqbridge.MQeMQBridgeQueue`
`mqbridge.MQeMQQMgrName`
`mqbridge.MQeMQQName`			

Message storage			

Table 12. Class optimization (continued)

Base		X	X
MQeMessageStoreException MQeAbstractMessageStore messagestore.MqeIndexEntry			
Standard	Add: Base	X	X
messagestore.MQeMessageStore			
Short filename. Always use 8.3 file name for messages.	Add: Standard		X
messagestore.MQeShortFilenameMessageStore			
4690 specific	Add: Short filename		
messagestore.MQe4690ShortFilenameMessageStore			
Message type			
Basic		X	X
Support for MQeMsgObject is in Mandatory classes			
MQSeries			
mqemqmessage.*			
Storage adapters			
Assured disk	Independence from OS lazy writes		X
adapters.MQeDiskFieldsAdapter			
Non-assured disk	Dependence on OS lazy writes Add: Assured disk		X
adapters.MQeReducedDiskFieldsAdapter			
Case-Insensitive	Add: Assured disk		X
adapters.MQeCaseInsensitiveAdapter			
Long to Short Filename Mapping			X
adapters.MQeMappingAdapter			
Midp RMS Storage	MIDP Only	X	
adapters.MQeMidpFieldsAdapter com.ibm.mqe.adapters.MQeMidpFieldsAdapter$RMSFile			
Memory	Volatile storage	X	X
adapters.MQeMemoryFieldsAdapter			
Administration			

Table 12. Class optimization (continued)

	Basic administration capability	Add: Local queue	X	X

		`MQeAdminMsg`
		`MQeAdminQueue`
		`MQeAdminQueue$1`
		`MQeAdminQueue$Timer`		

	Manage queue manager	Add: Basic administration capability	X	X
		`administration.MQeQueueManagerAdminMsg`		
	Manage connection definitions	Add: Basic administration capability	X	X
		`administration.MQeConnectionAdminMsg`		
	Manage communications listeners	Add: Basic administration capability	X	X
		`administration.MQeCommunicationsListenerAdminMsg`		
	Manage local queue	Add: Basic administration capability	X	X
		`administration.MQeQueueAdminMsg`		
	Manage administration queue	Add: Manage local queue	X	X
		`administration.MQeAdminQueueAdminMsg`		
	Manage remote queue	Add: Manage local queue	X	X
		`administration.MQeRemoteQueueAdminMsg`		
	Manage home server queue	Add: Manage remote queue	X	X
		`administration.MQeHomeServerQueueAdminMsg`		
	Manage store and forward queue	Add: Manage remote queue	X	X
		`administration.MQeStoreAndForwardQueueAdminMsg`		
	Manage bridge queue	Add: Manage remote queue		X
		`mqbridge.MQeMQBridgeQueueAdminMsg` `mqbridge.MQeCharacteristicLabels`		
	Manage a bridge to MQSeries	Add: Remote queues		

		`mqbridge.*AdminMsg`
		`mqbridge.MqeCharacteristicLabels`
		`mqbridge.MqeRunState`
		`mqbridge.MqeBridgeServices`
		`mqbridge.MQeBridgeExceptionCodes`		

Queue manager creation and deletion		`MQeQueueManagerConfigure`	X	X
Authenticators				
	mini-certificate			X

		`attributes.DHk (source may be generated)`
		`attributes.MQeSharedKey`
		`attributes.MQeRandom`
		`attributes.MQeWTLSCertificate`
		`attributes.MQeWTLSCertAuthenticator`		

Compressors				

Table 12. Class optimization (continued)

	GZIP	attributes.MQeGZIPCompressor		X
	LZW	attributes.MQeLZWCompressor attributes.MQeLZWDictionaryItem	X	X
	RLE	attributes.MQeRleCompressor	X	X
Cryptors				
	triple DES	attributes.MQe3DESCryptor		X
	DES	attributes.MQe3DESCryptor		X
	MARS	attributes.MQeDESCryptor		X
	RC4	attributes.MQeRC4Cryptor		X
	RC6	attributes.MQeRC6Cryptor		X
	XOR	attributes.MQeXorCryptor	X	X
Application security services				
	Local security	Add required: Cryptors	X	X
		attributes.MQeLocalSecure		
	Message-level security	Add required: Cryptors		X
		attributes.MQeMAttribute		
	Message-level security with digital signature & validation	Add: Public registry. Add required: Cryptors		X
		attributes.MQeMTrustAttribute		
Trace				

Table 12. Class optimization (continued)

	Collect binary trace in J2SE/CDC			X
		trace.MQeTraceToBinary trace.MQeTraceToBinaryFile		
	Collect binary trace to Midp RMS Store And or send to MIDP Trace servlet		X	
		trace.MQeTraceToBinary trace.MQeTraceToBinaryMidp		
	Base trace renderer			X
		trace.MQeTracePoint trace.MQeTracePointGroup trace.MQeTraceRenderer		
	Decode a binary file to readable form	Add: Base trace renderer		X
		trace.MQeTraceToReadable trace.MQeTraceFromBinaryFile		
	Trace to a readable output stream	Add: Base trace renderer		X
		trace.MqeTraceToReadable		
	Servlet collection of Midp binary trace	Add Base trace renderer		
		trace.MQeTraceToReadable examples.trace.MQeServlet		
Miscellaneous				
	Cryptographic support	Application or installation use only		X
		attributes.MQeCL (footnote?) attributes.MQeGenDH (generates a version of attributes.MQeDHk.java)		
	Mini-certificate server SupportPac ES03	MQe_MiniCertServer (or command line tool) See ES03 installation instructions		
	MQe_Explorer SupportPac ES02	MQe_Explorer See ES02 installation instructions		
Bindings		Access to Java classes from other languages		
	C language	bindings.*		
JMS		Support for the Java Message Service API		XX
		jms.* transaction.*		
User-defined MQe extensions				
		Authenticators Communications adapters Compressors Cryptors Logging classes Message classes Rule classes Security control Storage adapters Trace handler		

JMS requirements

In order to use the MQe JMS programming interface, the JMS interface classes are required.

These are contained typically in jms.jar.

MQe does not ship with jms.jar, and this must be downloaded before JMS can be used.

At the time of writing, this can be freely downloaded from http://java.sun.com/products/jms/docs.html. The JMS Version 1.0.2b jar file is required.

JNDI

In addition, if JMS administered objects are to be stored and retrieved using the Java Naming and Directory Interface (JNDI), the javax.naming.* classes must be available.

If Java 1 is being used, for example, a 1.1.8 JRE, jndi.jar must be obtained and added to the classpath.

If Java 2 is being used, for example a 1.2 or later JRE, the JRE might contain these classes.

You can use MQe without JNDI, but at the cost of a small degree of provider dependence. MQe-specific classes must be used for the ConnectionFactory and Destination objects.

You can download JNDI jar files from http://java.sun.com/products/jndi

MQe classes for Java requirements

To use the MQ bridge the *MQ Classes for Java* are required, version 5.1 or later.

These are packaged with MQ 5.3 and above. If using an earlier version of MQ then they are available for free download from the Web as SupportPac MA88, see MQe SupportPacs for web addresses.

For an example of how to setup the classpath to include MQ jar files, see batch files:

- <MqeInstallDir>\Java\Demo\Windows\javaenv.bat
- <MqeInstallDir>\Java\Demo\UNIX\javaenv

Occasionally, the jar files change between versions of MQ - if problems are encountered as a consequence of this, consult the documentation for MQ classes in order to determine the correct jar files to use.

Using WSDD smart linker

The smart linker tool that ships with WSDD (WebSphere Studio Device Developer) is used in the process of building and packaging an application into a jar or jxe file. The smart linker can remove classes (and methods) that are deemed not to be required; this can cause the removal of classes that are needed but dynamically loaded. MQe makes use of dynamic loading so care should be taken to either avoid this feature or to explicitly name classes that must be present, even though not explicitly referenced in the code.

To prevent unused classes being removed use the `-noRemoveUnused` option.

Otherwise, if the `-removeUnused` option is set then any class that is dynamically loaded must be specifically included. One option that can be used to achieve this is `-includeWholeClass`.

For example

```
-includeWholeClass "com.ibm.mqe.adapters.*"
```

will include all classes in the adapters package, and

```
-includeWholeClass "com.ibm.mqe.adapters.MQeTcpipHttpAdapter"
```

will include only the http adapter.

Multiple include (or exclude) options can be specified in the smart linker options file.

The following guidelines can be used to determine which classes are dynamically loaded. The basic guideline is any class that is referenced though an MQe class alias or any class that is set as a parameter when administering MQe resources will be dynamically loaded. This includes:

- Communications adapters
- Storage adapters
- Message stores
- Rules
- Aliases
- Cryptors
- Compressors
- Authenticators
- Queues
- Transporter
- Connection (refer to the following example)

An example of a set of includes needed for a simple MIDP application is:

```
-includeWholeClass "com.ibm.mqe.MQeQueue"
-includeWholeClass "com.ibm.mqe.MQeRemoteQueue"
-includeWholeClass "com.ibm.mqe.MQeHomeServerQueue"
-includeWholeClass "com.ibm.mqe.MQeTransporter"
-includeWholeClass "com.ibm.mqe.communications.MQeConnectionDefinition"
-includeWholeClass "com.ibm.mqe.adapters.MQeMidpFieldsAdapter"
-includeWholeClass "com.ibm.mqe.adapters.MQeMidpHttpAdapter"
-includeWholeClass "com.ibm.mqe.messagestore.MQeMessageStore"
-includeWholeClass "com.ibm.mqe.registry.MQeFileSession"
```

J2ME Midp specifics

When deploying the Java Application for the Midp environment a few additional comments are worth mentioning.

- The developer must use the Midp specific Storage and Communication adapters (see “Using WSDD smart linker” on page 266) and exclude any classes that are not Midp compliant.
- You can either use the prepackaged MQeMidp.jar file or your own reduced version, however a JAD file (Java application descriptor) must also be included detailing the Midlets available within the application. When deploying to the

device all classes should be packaged and preverified in one jar before deploying. However, whilst testing using an emulator several jars can be used by including them in the classpath
- Sun and IBM also provide tools that will generate the required .prc file for Palm Devices. See documentation within either Sun's Wireless Toolkit or IBM's WebSphere Studio Device Developer
- Care must be taken to ensure that all the required classes are included in either the jar/prc file or other executable. Some classes are dynamically loaded and may be missed when using any Smart-Linker. See "Using WSDD smart linker" on page 266 for more details.

4690 specifics

Take the following requirements into account when configuring MQe for use with 4690.
- Terminal Applications are restricted to 24 character maximum path length, but Store Controller Applications can have 127 characters. Java Applications are also restricted to the 24 length.
- The virtual file system (VFS) cannot hold greater than 64,000 files. With GB disk sizes being used, the C: drive may not have a limit on the number of files, depending on your operating system.
- When you want to access a file, you must specify the path that leads to it. The path consists of directory names that are separated by a backslash character ″\″ or a forward slash ″/″.

  Note: Although your system accepts both the ″\″ and the ″/″ character, it is probably less confusing to use one or the other.
- Examples elsewhere in this manual demonstrate how to configure your queue manager such that the data describing its resources, certificates, and other configuration data is stored in files with long filenames. These filenames are for a single top-level directory, which can also be located on the VFS drive namespace.
- Using the 8.3 format, the total character length of the fully-qualified filename exceeds the allowable limits imposed by the 4960 native file system. Therefore, in VFS :
  - The maximum length of a filename is 256 characters.
  - The maximum path length, including directories and files, is 260 characters.
  - The maximum directory depth is 60 levels including the root directory.
- MQe classes can be stored in long format names in VFS. However, for performance and convenience, as there are lots of class files, we would recommend that the application and MQe classes are packaged into a .jar files and deployed.
- According to the VFS manual ″The operating system provides support for file names greater than eight characters in length through the use of a 4690 Virtual File System (VFS)″.
- The VFS manual states: ″The VFS drive setting must be enabled through system configuration. On enabling VFS drive settings, the operating system creates two logical drives. C: and D:. The drive determines where the VFS directory is located. However, the information is actually stored on drives C: and D:. Drive M: information is stored on drive C:, and drive N: information is stored on drive D:. Once you have enabled VFS, you can use drives M: and N: to provide long file name support locally.″
- It is recommended that you use the `MQeCaseInsensitiveDiskAdapter` on the 4690 OS. This class implements a disk adapter that is insensitive to the case (upper or

lower) of the filename used during matching. Some JVM or OS combinations list files with different case to that in which they were created. This means that the simple filtering in the superclass ignores them. However this class converts both the comparator and the comparand to lowercase before performing the comparison. This ensures the best chance of finding a valid match. Note that the conversion to lower case may be inappropriate on platforms where the case is honoured, and where there are non-MQe files stored that could be confused by case. In summary, this adapter is suited for use with the 4690 filesystem.

Packaging

Following is a list of some of the techniques and tools that can be used to package applications ready for deployment to a device. The list is not a full list and does not go into any detail but is intended to provide an introduction to some of the ways a Java application can be packaged.

Single Jar file

Build a self-contained application with MQe embedded in it. This option minimizes the footprint and ensures that the classpath is kept to a minimum.

Multiple Jar files

Put the application into one jar file, and then also use either the supplied MQe jar files or construct a separate MQe jar file. Keeping MQe in one or more separate jars makes it easy to use MQe from multiple independent applications.

JNLP

JNLP (Java Network Launching Protocol and API) is an emerging standard for use in packaging and deploying Java applications. It is designed to automate the deployment, via the web, for applications written for the J2SE platform.

OSGi

OSGi or Open Services Gateway Initiative defines define a platform for the packaging of and dynamic delivery of Java software services to networked devices. This is achieved via a consistent, component-based, architecture for the development and delivery of Java software components known as bundles and services. Both MQe components and applications can be turned into OSGi bundles and services for use in an OSGi environment. The bundles are delivered from a bundle server There are several products that provide bundle servers together with the client code to handle the installation and lifecycle of bundles. Depending on implementation the bundles can be downloaded on demand, and updated automatically when a new version is available. IBM WSDD (WebSphere Studio Device Developer) ships with SMF (Service Management Framework), which assists in the creation and testing of bundles together with a bundle server.

See more at “Open Services Gateway initiative (OSGi)” on page 272.

Midlet

An MQe J2ME MIDP application must be packaged as a midlet or midlet suite (.jad and .jar).

Palm specific

In order to run on a Palm device a Java application must to be packaged in a prc file, which is a Palm specific format. The IBM WebSphere Studio Device Developer product ships with a tool that will package a Java application as a prc file.

JXE

IBM WebSphere Studio Device Developer has a SmartLinker tool that can produce an optimized packaging of an application that contains the minimum set of required classes and methods for the deployment platform. The output from the smartlinker is stored in a .JXE file which is understood by the IBM j9 Java runtime.

Installer

There are several tools that will package an application ready for installation on one or more platforms. A couple of examples of these are InstallShield and self extracting zip files.

Roll your own distribution mechanism

For instance using a Java class loader that can load classes over a network.

Deployment to devices

Following is a list of some of the techniques and tools that can be used to deploy applications to devices. The list is by no means complete and does not go into any detail but is intended to provide an introduction to some of the ways a Java application can be deployed.

Device specific tools

Most devices ship with tools that allow applications to be copied across and installed. For instance:

- ActiveSync for PocketPC
- Hotsync for Palm

Development tools

Many IDEs (Integrated Development Environments) such as IBM WSDD (WebSphere Studio Device Developer) provide tools that allow deployment of applications onto a device and debugging of the application from the development environment.

OSGi related management

OSGi or Open Services Gateway Initiative defines define a platform for the packaging of and dynamic delivery of Java software services to networked devices. This is achieved via a consistent, component-based, architecture for the development and delivery of Java software components known as bundles and services. Both MQe components and applications can be turned into OSGi bundles and services for use in an OSGi environment. The bundles are delivered from a bundle server. There are several products that provide bundle servers together with the client code to handle the installation and lifecycle of bundles. Depending on implementation the bundles can be downloaded on demand, and updated automatically when a new version is available. IBM WSDD (WebSphere Studio Device Developer) ships with SMF (Service Management Framework), which assists in the creation and testing of bundles together with a bundle server.

See more at “Open Services Gateway initiative (OSGi)” on page 272.

JNLP

JNLP or Java Network Launching Protocol and API, is an emerging standard, for use in packaging and deploying Java applications. It is designed to automate the deployment, via the web, for applications written to the J2SE platform.

Device management products

There are several products on the market that can be used for large-scale deployment of software. One example is Tivoli® Configuration Manager from IBM.

C deployment

Supplied DLLs

To deploy applications on the PocketPC 2000, 2002 and 2003 devices, you must copy the MQe DLLs to the device. Copy the DLLs to the Windows directory, the root directory, or the same directory that holds the application. The following list shows which DLLs you need for different MQe entities:

For the local queuing base

- HMQ_Core.dll
- HMQ_DiskAdapter.dll
- HMQ_HAL.dll
- HMQ_nativeAPI.dll
- HMQ_nativeOSA.dll
- HMQ_RegistryFileSession.dll
- HMQ_LocalQueue.dll

Along with the base DLLs you require the following DLLs depending on how you wish to configure your application:

Remote queuing

HMQ_HttpAdapter.dll

Note: You can remove HMQ_LocalQueue.dll, if you do not want to support administration queues or local queueing.

Synchronous remote queue support

HMQ_SyncRemoteQueue.dll

Asynchronous remote queue support

HMQ_AsyncRemoteQueue.dll

Home server queue support

HMQ_HomeServerQueue.dll

Administration queue support

HMQ_AdminQueue.dll and HMQ_LocalQueue.dll

RLE compressor support

HMQ_RleCompressor.dll

RC4 crytpor support

HMQ_RC4Cryptor.dll

Support for included examples

BrokerDLL.dll

Open Services Gateway initiative (OSGi)

Open Services Gateway initiative (OSGi) is an application framework capable of deploying java applications or services, which can be dynamically employed, updated, or removed. Therefore, it can be a very useful means for providing service updates and ensuring that all the required classes for an application are made available as and when required. MQe provides an example bundle that provides MQe messaging within this framework.

The following topics explain more.

MQe example bundle contents

MQe provides one main bundle for OSGi development and two example application bundles that provide hints on how to create an MQe client or server application within OSGi. No bundle exports or imports a service; they all rely on package dependency. The following table details the bundles and their dependencies.

Table 13. Bundles and dependencies

Bundle name	Description	Export packages	Import packages
MQeBundle.jar	Bundle containing all the required MQe classes excluding mqbridge functionality	com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.attributes com.ibm.mqe.communications com.ibm.mqe.messagestore com.ibm.mqe.mqemqmessage com.ibm.mqe.registry com.ibm.mqe.trace	
MQeServerBundle.jar	Example bundle containing an MQe Server application		com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.trace org.osgi.framework
MQeClientBundle.jar	Example bundle containing an MQe Client application		com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.trace org.osgi.framework

Both example application bundles, MQeClientBundle.jar and MQeServerBundle.jar contain bundle activators which start and stop the application when the framework starts or stops the bundle. The bundles are in MQE_HOME/Java/Jars.

Using MQe within OSGi

When developing your own bundles, importing the correct MQe packages into your bundles manifest file ensures that the MQe bundle is also installed into the framework when your bundle is installed.

One major factor in developing a bundle is that only one MQe queue manager can be run within an OSGi runtime. This means that there may be conflicts if several bundles are installed and each requires its own queue manager. Careful design of

the bundle application is required to eliminate this problem. However, there should be no limit on the number of bundles that can use the same queue manager.

Running the example bundles

As an example of how to use MQe within the OSGi environment, we provide two example application bundles that are designed to work together in a simple scenario.

The scenario consists of a client application and a server application:-

- The Server simply sits and waits for messages and prints out any that it receives,
- The Client just sends one message.

Within this scenario it is possible to have multiple Clients sending to the same Server or the same Client can be stopped and restarted to send another message to the Server.

These bundles are explained in more detail in the following topics.

Server application (MQeServerBundle.jar)

When this bundle is started, an MQeQueueManager is created and started, with a listener and default queues in memory.

The Application code is then run in a new thread and waits for incoming messages using a message listener; any received messages are displayed in the console. This thread continues to listen until the bundle is stopped, at which time it stops and then deletes the MQeQueueManager.

Client application (MQeClientBundle.jar)

When this bundle is started it checks to see if an MQeQueueManager is already running in the JVM, and if so, it assumes it is running in the same runtime as the server, and so uses that queue manager. If no queue manager is detected then a new one is defined and started in memory and a connection definition and remote queue definition are setup to the server.

Client application code is then run in a new thread which sends a single message to the server. No checks are made to ensure the message is received.

When the bundle is stopped, if a new QueueManager was created for the Client, it is stopped and deleted.

The source for the classes included in the bundles can be seen in the MQe\Java\examples\osgi directory. More details are given in the Java API Programming Reference for these classes.

Some points to note when running the applications:

- Each application was written with two parts in mind. The first is setup of the underlying MQe messaging infrastructure, and the second is the main application. This is why each one has a separate class providing function for each part.
- The MQeClientBundle.class and MQeServerBundle.class are both started in their own threads by the bundle activator start method. This way the start method is

not delayed in completing as the tasks of sending and receiving messages can take some time. This ensures a smooth transition of the bundles state from resolved to started.

Note: The Client and Server share the same MQeAdmin class in their bundles. This class could have been placed in its own bundle to avoid the duplication but for simplicities sake we have not done this.

- The Server must always be started before any Clients. Each Server must run in its own runtime. A single client can share the server's runtime or can reside in its own.

Running the example

Whichever way you run the examples, the MQeBundle.jar bundle is required by both the client and server and must be present on the Bundle Server.

To run the example, first start the Server:

1. Import the MQeServerBundle.jar bundle onto the Bundle Server.
2. Start a new SMF (Service Management Framework) runtime, and install and start the MQeServerBundle bundle on it. This should also install the three prerequisite bundles.
3. The server then starts listening, you should see output on the console including:

   ```
   'MQeServerBundle - registering message listener'
   ```

   This means the server is ready for messages.

Next you need to run a client to send a message. There are two methods for runnning the client bundle:

Method 1

In the same SMF runtime as the server:

1. Import the MQeClientBundle.jar bundle onto the Bundle Server.
2. Install and start the MQeClientBundle bundle on the runtime.
3. The client now starts and sends a message, which the server will print on the console. You can stop and start the client bundle to send another message.

Method 2

In separate SMF runtimes:

1. Import the MQeClientBundle.jar bundle onto the Bundle Server.
2. Start a new SMF runtime, and install and start the MQeClientBundle bundle on it. This should also install the three prerequisite bundles.
3. The client starts and sends a message, which the server will print on the console. You can stop and start the client bundle to send another message.

By default the example expects both client and server to be on the same machine running with the receiver listening on port 8085. However, you can change the port and address of the server, that is run the server on a separate machine. Before the server is started, tell it which port to run on by setting the java system property, `examples.osgi.server.port`. This can be set in the Runtime IDE by selecting **Show runtime properties** from the drop down menu.

To tell the client the address and port that the server is listening on, before starting the client set the system properties `examples.osgi.server.address` and `examples.osgi.server.port`.

Note: The server ignores the address property if it is not present. Also, if the client has already been run and you want to change the address and port, the runtime needs to be terminated and restarted to ensure that old MQeConnectionDefinition information is wiped from memory.

Providing user-defined rules and dynamic class loading

The OSGi runtime controls package visibility across bundles. If a bundle does not explicitly import a package, then it will not have access to classes within that package when it comes to dynamically loading them. This is especially important to MQe, because it has been designed with this flexibility in mind. Without some small changes to the bundles, developers cannot use 3rd party or their own Rules or Adapters. There are two ways to remove this problem:

1. OSGi version 3 includes a DynamicImport-Package statement for the bundles manifest file. This has been included in the MQeBundle.jar and when the user-defined class's package is exported from its bundles manifest, MQe will be able to have access to this class.

   Note: This functionality is available to SMF version 3.1.0 or higher.
2. Create a new MQeLoader and add all the user-defined classes before they are used, most likely within the bundles activator, for example:

   ```
   String MyRule = "UserQMRule";
   MQeLoader loader = new MQeLoader();
   loader.addClass(MyRule, Class.forName(MyRule));
   MQe.setLoader(loader);
   ```

   Note: Take care that the loader within MQe is not replaced with another loader from another bundle during the application runtime.

Problem solving

Problem determination

Before you start problem determination in detail, it is worth considering whether there is an obvious cause of the problem, or an area of investigation likely to give good results. This approach to diagnosis can often save a lot of work by highlighting a simple error, or by narrowing down the range of possibilities.

The checklist below raises some fundamental questions that you need to consider.

As you go through the list, make a note of anything that might be relevant to the problem. Even if your observations do not suggest a cause straight away, they could be useful later if you have to carry out a systematic problem determination exercise.

The preliminary checks that you should make are as follows:-

- Has MQe run successfully before?
- Are any appropriate environment variables set correctly?
- Are there any error messages, return codes, or exceptions, that explain the problem?
- Can you reproduce the problem?
- Have you made any changes since the last successful run?
- Is there a problem with the network?
- Does the problem affect all users?
- Have you applied any service updates?

Common problems

Client is unable to connect to server - it appears to hang

- Check that the communications protocol being used by the client matches the protocol being used by the server.

Messages are stuck

- Ensure that the channel timeout settings are appropriate for the network.
- Ensure that the adapter timeout settings are appropriate for the network

Home server queue stops pulling messages from the store queue

- The home server queue has a polling interval that may be set. However if the thread polling the queue terminates unexpectedly, no exception is thrown to the application. An alternative approach is to set the poll interval to zero, and implement a queue manager rule to loop on a call to trigger transmission.

Asynchronous remote queue does not send message in a timely manner

- By default an asynchronous remote queue will attempt to send a message when it is put to the queue. If the background thread is unsuccessful for any reason, then no attempt is made to resend the message until another message is placed on the queue. To avoid this, implement a queue manager rule to loop on a call to trigger transmission.

© Copyright IBM Corp. 2000, 2004

Cannot create bridge definitions

- Check that the MQe bridge classes are on the CLASSPATH
- Check that the MQ Java classes are on the CLASSPATH

Tracing and logging

Tracepoints generated from MQe

All of the MQe trace points use negative trace point numbers. They are provided to facilitate problem diagnosis for the IBM Service team, when investigating a reported problem on the MQe product.

Each trace point may change its meaning, value, and sequential position, between versions of the MQe classes. A trace point used in one version of MQe might never be issued in another. For this reason, we strongly recommend that in your applications you do not use a trace point as a trigger for application logic.

When rendering trace point information to a readable format, maintain a consistent version between all of the MQe classes. Failure to do so might result in misleading information being written to the trace output.

Tracing and logging with Java

The trace mechanism provided and used by MQe has the following features:

- A pluggable interface to allow user-written trace handlers to be implemented if required.
- A variety of implementations of the trace handler interface to cater for a variety of uses. One such implementation supports a crude form of circular logging, so older trace information is discarded when newer trace information becomes available. See the com.ibm.mqe.trace.MQeTraceToBinaryFile for more details.
- A separation between the trace point number, and the meaning, or textual representation of that trace point. This separation of the number from lengthy meaningful string information allows for collection of the trace point numbers to be performed at runtime, and the rendering of that information to a readable format to be done offline at a later time. This can mean trace information files are smaller and generated more quickly at the point of capture, but much larger and more accessible at the time they are read.
- Dynamic, runtime filtering of trace information.

Generating trace information (Java)

Tracing in the Java codebase is performed using the `com.ibm.mqe.MQeTrace class`. All calls to `com.ibm.mqe.MQeTrace.trace()` methods pass the following information:

- A number, by which the trace point can be identified.
- A group bit-mask, which identifies this trace point as being classified as part of one or more groups of trace points. This information is used in conjunction with the `MQeTrace.setFilter()` method, to allow unwanted trace information to be filtered out at runtime. Many of the bits in the group bit-mask have a defined meaning. For example, if the `MQeTrace.GROUP_ERROR` bit is set, then the trace point indicates that an error is being reported. Multiple group bits can be set for the same trace point.
- A number of parameters. A `tostring()` method is invoked for each parameter, so that a string is extracted at runtime, and added to the trace point.

Classes shipped in MQe generate lots of trace information using these methods, such that the trace point numbers are all negative. We recommend that programs using this trace mechanism use positive numbers, or zero.

Several bit-fields are reserved for user applications, for example, the MQeTrace.GROUP_MASK_USER_DEFINED bit-fields. For convenience, MQeTrace.GROUP_USER_DEFINED_1 maps to one such bit, for example:

```
MQeTrace.trace(this, (short) 1, MQeTrace.GROUP_ERROR |
                                MQeTrace.GROUP_USER_DEFINED_1, thingToLog );
```

This statement implements a logical AND operation on the GROUP_ERROR and GROUP_USER_DEFINED_1, maintaining the runtime filter with the MQeTrace class. If the result is non-zero, then the corresponding method on the MQeTraceHandler interface class is called, if a handler has been set.

There are several variants of the MQeTrace.trace() method, including methods that trace different numbers of parameters with the trace point.

Capturing trace information (Java)

MQe does not automatically capture the trace information provided by the MQeTrace.trace() methods. The solution programmer must capture the trace messages. We strongly recommend that your solution includes a mechanism to allow the capture of MQe trace events, as this output may be requested by the IBM service teams when investigating any problems reported.

To capture MQe trace information, you need to ensure that

- A trace handler has been provided, and set using the MQeTrace.setHandler() method.
- The runtime filter maintained by the MQeTrace.setFilter() method is not excluding the information you want to capture.

The required trace handler must implement the MQeTraceHandler interface. MQe ships with several trace handlers, used for different purposes:

MQeTraceToReadable
: This renders trace information to a printstream in a readable format.

MQeTraceToBinaryFile
: This collects trace information into a file, or sequence of files.

MQeTraceFromBinaryFile
: You can use this to render this binary information file format into readable text.

MQeTraceToBinaryMidp
: Collects binary trace information when running inside a MIDP Java environment.

```
// Allocate a trace instance, so that our handler
// is not garbage collected when it's on.
myMQeTrace = new MQeTrace();

// Allocate a trace handler
// This one puts trace output to stdout by default.
MQeTraceHandler handler = (MQeTraceHandler)
new com.ibm.mqe.trace.MQeTraceToReadable();

// Set this handler as the one MQe uses.
MQeTrace.setHandler(handler);
```

```
// Set the filter so we collect those
// pieces of trace we are interrested in.
// In this case, collect all the default trace information.
MQeTrace.setFilter(MQeTrace.GROUP_MASK_DEFAULT);

...

// To end trace set the filter to zero and the handler to null
MQeTrace.setFilter(0);
MQeTrace.setHandler(null);
```

This example shows the creation of a trace handler,`MQeTraceToReadable` in this case, and setting of the filter to capture the default trace information. This can result in lots of information being captured. You can use a more restrictive filter to capture only a subset of the data. For example, collecting errors, warnings, and user-coded trace points might be more appropriate:

```
MQeTrace.GROUP_ERROR | MQeTrace.GROUP_WARNING | MQeTrace.GROUP_MASK_USER_DEFINED
```

Note:

1. The IBM Service team may ask you to use the `MQeTrace.GROUP_MASK_ALL` value when diagnosing a problem.
2. When using the MQeTraceToBinaryMidp tracehandler, you require an additional step to recover the trace. The MIDP tracehandler either stores the trace in a record store or in memory. Once trace has finished, call `sendDataToUrl()` to recover this binary data. By default, this sends the data to a servlet. For more information, refer to the `examples.trace.MQeTraceServlet` section of the MQe Java Programming Reference.

Writing your own trace handler (Java)

Solution providers may wish to write their own trace handlers, to

- Do more complex filtering.
- Store trace information in a different place or form to that used by the supplied trace handlers
- Reroute trace information generated through this mechanism to another trace capture mechanism. For example, the trace handlers supplied with MQe rely on function supplied by underlying classes:

  com.ibm.mqe.trace.MQeTracePointGroup
  This class holds information about a logical grouping of trace points.

  com.ibm.mqe.trace.MQeTraceRenderer
  Provides a programmatic way of managing a collection of tracePointGroups and tracePoints information. It provides methods to add or remove tracePointGroups, individual tracePoints to and from the collection of tracePointGroups, and collection of tracePoints.

  com.ibm.mqe.trace.MQeTracePoint
  A collection of information that describes a particular trace point.

The trace handlers in the product populate a series of MQeTracePointGroup objects with a collection of MQeTracePoint objects. The groups are added to the MQeTraceRenderer, and the MQeTraceRenderer is used to map from the trace point number passed on the `MQeTrace.trace()` methods, to a readable string.

The separation of the readable string from the trace point number allows the code to collect just the number, and separate the information collection stage from the stage that renders to readable strings.

Where possible, the trace handlers supplied also gather stack trace information when a java.lang.throwable is passed as a parameter to the `MQeTrace.trace()` method.

You can implement the trace handler interface, and intercept trace information from your application and the MQe system classes. For examples of this, refer to the following classes in the MQe Java Programming Reference:

- `examples.trace.MQeTrace`
- `examples.trace.MQeTraceToFile`

You can add trace points to existing trace point groups, or to your own trace point groups. You can add these to the base MQeTraceRenderer, and use them in conjunction with the existing trace handlers. For an example of this, please refer to the MQeTrace class section of the MQe Java Programming Reference.

Tracing and logging with C

This section describes trace and logging in the C codebase. It shows you how to enable MQe trace on PocketPC and PocketPC2002 devices and emulators.

Trace architecture (C)

In MQe, trace is configured globally. This means that trace is enabled for the device, and when enabled, all C MQe applications generate trace.

You can configure the location where you want trace files to be written.

Each application generates a unique file in the form MQEnnnnn.trc, where ″nnnnn″ is the process identifier of the application.

MQe trace files are written in a binary format to minimize their size.

You can either send these binary trace files directly to an IBM Service Representative to decode them or, alternatively, use the MQenativeTraceFormatter.exe utility provided with MQe. This utility runs on Windows, but does not run on PocketPC. It takes the trace file as an argument and prints the decoded output to standard out. The output can be captured in a file by running a command, such as:

```
MQenativeTraceFormatter.exe AMQ12345.trc > AMQ12345.txt
```

This will decode the file AMQ12345.trc and place the output in a file called AMQ12345.txt.

Configuring trace (C)

Trace is controlled on the PocketPC via entries in the Windows Registry. These trace values are under the HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQe\CurrentVersion\Trace key. You can set the values in several ways:

- Manually, using a registry editor, such as the Remote Registry Editor provided with eMbedded Visual Tools V3.0
- With a .Reg file, which you can download to the device and then execute
- Programatically, using the supplied mqeTrace_setOptions function

For information on the mqeTrace_setOptions function, see the C Programming Reference on the product CD. If you set the value manually or use a .Reg file, all values should be of type REG_SZ. MQe supports the following values:

Table 14. Trace values supported in MQe

Value Name	Supported values	Description
Enable	Yes or no	Turns trace on and off.
Location	Full path	Directory where trace files are written to. The location string must be a valid file path, for example mqetrace.
Timestamp	Yes or no	Determine if timestamp information is added to each tracepoint. Set to ″no″ to reduce file size and increase speed.
Parameters	Yes or no	Determine if parameter information is added to each tracepoint. Set to ″no″ to reduce file size and increase speed.
WrapLength	Value	Advanced value, described in the following list under Wraplength.
SubtractMethodFilter	Value	Advanced value, described in the following list under AddMethodFilter and SubtractMethodFilter.
AddMethodFilter	Value	Advanced value, describted in the following list under AddMethodFilter and SubtractMethodFilter.

In the table, the following conditions apply:

WrapLength

This is the maximum size, in bytes, that an individual trace file will reach. Once this value is reached, the trace file begins to wrap using a ″circular buffer″ algorithm. However, this takes a considerable amount of time, and may significantly slow down execution speed once the file starts wrapping. Therefore, leave this value at -1, except in circumstances where disk space is at a premium.

Note: This is the maximum length of a single trace file. If an application is run multiple times, or multiple applications are run, then each generated trace file reaches this size.

AddMethodFilter and SubtractMethodFilter

These values allow sophisticated control over exactly what trace points are produced. Incorrect use can seriously limit the effectiveness and understandability of the trace files. You should leave these fields blank, unless an IBM service representative instructs you otherwise. If you do send trace files to IBM, you must include details of what both of these fields are set to.

MQe Diagnostic tool

MQe includes a small diagnostic tool, `MQeDiagnostics`, that can be used to gather the information required by technical support personnel to assist with problem determination. The tool collects information about the local MQe environment. In particular:

- CLASSPATH and PATH information
- Java and C system variables

- Version information of the MQe classes

No personal information or MQe message data is collected by this program, and it should normally only be used at the request of IBM technical support personnel.

This tool should not be confused with the trace facility, which is used to gather debugging information on a running MQe system.

Windows diagnostics

1. From a command prompt change to the <mqe_install_dir>\mqe\Java\demo\Windows\ folder.
2. Edit the MQeDiagnostics.bat file to suit your environment. The file makes use of the JavaEnv.bat script, so either ensure that JavaEnv.bat correctly sets up your *CLASSPATH* and *PATH* environment variables, or set them up in the MQeDiagnostics.bat script.
3. Run the MQeDiagnostics.bat file and follow the on screen prompts.
4. Once the tool has completed, look through the MQeDiagnostics.out file for any errors. Common errors include:

   ″.\MQeDiagnostics.properties could not be found″

   The tool requires the MQeDiagnostics.properties file to be supplied as input. Edit MQeDiagnostics.bat so that it points to the correct location for this file and rerun the tool.

   ″com.ibm.mqe.support.MQeDiagnostics is not recognized as an internal or external command...″

   JavaEnv.bat is not configured correctly. Edit MQeDiagnostics.bat and JavaEnv.bat if necessary and rerun the tool.

   ″java.lang.NoclassDefFoundError: com/ibm/mqe/support/MQeDiagnostics″

   Edit JavaEnv.bat and MQeDiagnostics.bat if necessary so that the <mqe_install_dir>\MQe\Java\Jars\MQeDiagnostics.jar can be found in the *CLASSPATH* environment variable.

   Note: Not all MQe classes can supply version information, so the MQeDiagnostics.out file may include some ″Unknown version!″ messages.
5. Send MQeDiagnostics.out to the MQe support personnel, as described in “Information required by IBM support” on page 284.

Unix diagnostics

1. From a command prompt change to the C folder name or the <mqe_install_dir>\mqe\Java\demo\UNIX\ folder.
2. Edit the MQeDiagnostics script to suit your environment. The file makes use of the JavaEnv script, so either ensure that JavaEnv correctly sets up your *CLASSPATH* and *PATH* environment variables, or configure them directly from within the MQeDiagnostics script.
3. Run the MQeDiagnostics script and follow the on screen prompts.
4. Once the tool has completed, look through the MQeDiagnostics.out file for any errors. Common errors include:

   ″.\MQeDiagnostics.properties could not be found″

The tool requires the MQeDiagnostics.properties file to be supplied as input. Edit MQeDiagnostics.bat so that it points to the correct location of this file and rerun the tool.

″com.ibm.mqe.support.MQeDiagnostics : command not found″

The file JavaEnv is not configured correctly. Edit the files MQeDiagnostics and JavaEnv if necessary and rerun the tool.

″java.lang.NoClassDefFoundError: com/ibm/mqe/support/MQeDiagnostics″

Edit the files JavaEnv and MQeDiagnostics if necessary so that the <mqe_install_dir>\MQe\Java\Jars\MQeDiagnostics.jar file can be found in the *CLASSPATH* environment variable.

Note: Not all MQe classes can supply version information, so the MQeDiagnostics.out file may include some ″Unknown version!″ messages.

5. Send MQeDiagnostics.out to the MQe support personnel, as described in "Information required by IBM support."

Other systems diagnostics

On other systems, the `MQeDiagnostics` tool should be invoked directly.

1. Add the MQeDiagnostics.jar file to your classpath.
2. Invoke the `com.ibm.mqe.support.MQeDiagnostics` class from the Java runtime environment. For example:

```
java com.ibm.mqe.support.MQeDiagnostics MQeDiagnostics.properties > MQeDiagnostics.out
```

   (The program takes the MQeDiagnostics.properties file as an argument. The output is redirected to a file).
3. Send MQeDiagnostics.out to the MQe support personnel, as described in "Information required by IBM support."

Information required by IBM support

If you cannot resolve problems that you find when you use MQe, or if you are directed to do so by an error message generated by MQe, you can request assistance from your IBM Support Center.

Before you contact your Support Center, use the checklist below to gather important information. Some items might not necessarily be relevant in every situation, but you should provide as much information as possible to enable the IBM Support Center to re-create your problem.

For your system:

- Operating system being used, and Version, Level and any fixpacks or fixes applied (On UNIX you can find the version installed by using the `uname -a` command).
- MQe Version, Level and any fixpacks or fixes applied
- JVM Version, Level and any fixpacks or fixes applied
- Any relevant software used by MQE application
- Codebase - Server: Java or C Bindings; Client: Java, C or C Bindings
- Message flow:-
  - Are messages pulled - using home server queue on client and store queue on server

- Are messages pushed - synchronous or asynchronous remote queues or forward queue
- Explicit use of trigger transmission
- Is the application applying assured delivery (using confirm id and put with putConfirm)
- Message expiry
- Is security involved
- Network being used - e.g. GSM, GPRS, LAN
  - Communications adapter
  - Communications settings - adapter (for example: timeout, retries, packet size) and Channel (timeout)
- Is a gateway to MQ involved
  - MQ Bindings or MQ Client
  - Message created using MQeMQMessageObject or MQeMsgObject
  - Transformer used on Bridge
- Rules
  - Where are they used
  - Purpose
- Use of tools
  - MQe_Explorer
  - MQe_Script
- Administration

For your problem:

- A concise description of the problem
- How to re-create the problem
- Any traces you can generate, as follows:-

Traces for MQe:

- The file MQeDiagnostics.out, generated as described in "MQe Diagnostic tool" on page 282. Create a .zip file using any zip utility.
- Appropriate trace files generated according to the advice in "Tracing and logging" on page 278. Create a .zip file using any zip utility.
- A sample of the messages being used when the problem arose.

Traces for MQ:

- All current trace and error logs, including relevant Windows Event log or UNIX platform syslog entries and FFST output files. You can find these files, which have the extension .fdc, in the errors subdirectory within the MQ home directory.

Programming reference

If you have installed the two extra reference plugins they appear in this section in the Contents.

The plugins and their contents are:-

API References
- Java API Programming Reference
- C API Programming Reference

C Programming Guides
- C Bindings Programming Guide
- C Programming Guide for Palm OS

This section also contains:-

JMX Attributes and operations

The topics in this section describe the attributes and operations that each JMX-instrumented WMQe resource can access.

Each resource is described here by up to three sub-topics:

1. **Attributes**

   In the table on these pages:
   - The **Attribute Name** is the name returned by the `Attribute class getName()` method or the `MBeanAttributeInfo getName()` method.
   - The **Attribute description** describes the attribute.
   - The **Attribute Type** is the String representation of the attribute data type retrieved using the `MBeanAttributeInfo getType()` method (This is referred to as the *class name literal* for the type in Data types).
   - The **Read/Write** column indicates whether an attribute is read-only (RO), or can also be updated (RW).
2. **Operations**

   These pages show:
   - The **Operation Name**
   - The **Operation description**
   - The **Parameter**. For more details on what each parameter is, see the following sub-topic **Operations parameters**, where present.

   Unless otherwise stated, the return type of all operations is `java.lang.Void`. To ensure that default values are used, leave parameter entries blank on a GUI interface or use a value of null in a programmatic interface.
3. **Operations parameters**

   These pages explain the parameters on the preceding page, showing:
   - **Parameter name**
   - **Parameter type**
   - **Parameter description**

© Copyright IBM Corp. 2000, 2004

Admin MBean

Attributes

Table 15. Admin MBean attributes

Attribute name	Attribute description	Attribute type	Read/Write
AdminMsgExpiry	The default expiry time (in milliseconds) for sent admin messages	java.lang.Integer	RW
AdminQName	The default name of the admin queue to use for sending admin messages	java.lang.String	RW
CacheInterval	The default time (in milliseconds) to retain cached values (see below)	java.lang.Long	RW
InquireOnConnect	Indicates whether remote queue manager resources are inquired upon and MBeans for these created when new connection definitions are created	java.lang.Boolean	RW
MsgPollInterval	The default time (in milliseconds) to wait for admin reply messages when performing remote admin	java.lang.Long	RW
LocalMsgTimeout	The default length of time (in milliseconds) to check for admin reply messages when performing local admin	java.lang.Long	RW
RemoteMsgTimeout	The default length of time (in milliseconds) to check for admin reply messages when performing remote admin	java.lang.Long	RW

Note:

1. In the current implementation, the values set for these MQeAdminJmx attributes are not persistent between deletion and re-creation of the MBean so must be reset whenever a new MQeAdminJmx MBean is created. This effectively means that remote queue managers must be refreshed so that their children are all visible via JMX.
2. The attribute *CacheInterval* relates to a caching mechanism employed in the WMQe JMX interface. Attribute values are cached for this duration and inquires during the cache interval do not result in a refresh of the values. Attribute values are refreshed whenever an update is done (that is to say, the value of one or more of the attributes is changed). In that instance, the cache clock is reset to zero and no refresh will take place until either the *CacheInterval*expires or another update is done. The default value is 0.
3. *InquireOnConnect* is used whenever new connection definitions are created or loaded. If this attribute has a value of true, an attempt will be made to inquire upon the remote queue manager accessed directly by this connection definition. If the inquire is successful, an attempt will be made to create MBeans for all of the remote queue manager child resources. If the attribute value is false, an MBean will be created for the remote queue manager but no MBeans will be created or registered for its child resources. The creation of the children MBeans can be effected

when desired by using the MQeRemoteQueueManagerJmx refresh() method. The default setting is false which means that no children MBeans for remote queue managers are created at start-up.

4. *MsgPollInterval* specifies how frequently the admin reply message should be looked for. The lower the number, the more times the reply message will be searched for during the *Local/RemoteMsgTimeout*period. This relates effectively to remote administration — for local administration, there should only need to be a single attempt to retrieve the reply message.
5. *LocalMsgTimeout* and *RemoteMsgTimeout*indicate the length of time that it takes to check for a reply message. If a reply message is not returned within the specified time, then the inquire/update returns in an unknown state. For all local administration, a reply message should always be received. A case when a reply message may not be received is if remote administration is taking place and the remote queue manager does not have a connection definition back to the originating queue manager. In this case setting the *RemoteMsgTimeout* value to zero may be useful as it is already known that a reply message will not be received. In every case where a reply message is not received an exception will always be thrown. Setting the *RemoteMsgTimeout* to zero does not change this.
6. *LocalMsgTimeout*, *RemoteMsgTimeout* and *MsgPollInterval* have defaults 10,000ms, 10,000ms and 10ms respectively. Resetting these values takes effect immediately and the new values are in force until they are reset again or the application is terminated.

Operations

There are no operations for this MBean

Queue manager

Attributes

Table 16. Queue Manager attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Aliases	Alternative names for the resource	[Ljava.lang.String;	RW
ChannelAttributeRules	The rule class (or alias) to be associated with the channel attribute	java.lang.String	RW
ChannelTimeout	The time (in milliseconds) after which an outgoing idle channel will be turned off	java.lang.Long	RW
Connections	A list of connections owned by this queue manager	[Ljava.lang.String;	RO
MQBridges	A list of bridges owned by this queue manager	[Ljava.lang.String;	RO
Queues	A list of queues owned by this queue manager	[Ljava.lang.String;	RO

Table 16. Queue Manager attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
Rule	The rule class (or alias) to be used by this queue manager	java.lang.String	RW
MaxTransThreads	The maximum number of threads that will be spawned to service the transmission needs of the queue manager	java.lang.Integer	RW
Version	The version of WMQe hosting the queue manager	[Ljava.lang.String;	RO
CommsListeners	A list of communications listeners owned by this queue manager	[Ljava.lang.String;	RO
BridgeCapable	Indicates whether the queue manager is bridge capable	java.lang.Boolean	RO
MQeClass	The class of the resource	java.lang.String	RO
QMsgStore	The default message store class (or alias) for a queue that determines how messages on that queue are stored	java.lang.String	RW
QAdapter	The default storage adapter class (or alias) for a queue	java.lang.String	RW
QPath	The default path for messages to be stored for queues	java.lang.String	RW

Operations

Table 17. Queue Manager operations

Operation name	Operation description	Parameter [1]
addAlias [2]	Adds an alias to the queue manager	alias
removeAlias [2]	Removes an alias from the queue manager	alias
addAlias	Adds an array of aliases to the queue manager	aliases
removeAlias	Removes an array of aliases from the queue manager	aliases
createAdminQueue	Creates a new admin queue	queueName, messageStore(optional), adapter(optional), path(optional)
createApplicationQueue	Creates a new application queue	queueName, messageStore(optional), adapter(optional), path(optional)
createHomeServerQueue	Creates a new home server queue	queueName, getFromQMgrName

Table 17. Queue Manager operations (continued)

Operation name	Operation description	Parameter [1]
createAsyncProxyQueue	Creates a new asynchronous proxy queue	queueName, destinationQMgrName, messageStore(optional), adapter(optional), path(optional)
createSyncProxyQueue	Creates a new synchronous proxy queue	queueName, destinationQMgrName
createStoreQueue	Creates a new store queue	queueName, messageStore(optional), adapter(optional), path(optional)
createForwardQueue	Creates a new forward queue	queueName, forwardToQMgrName, messageStore(optional), adapter(optional), path(optional)
createCommsListener	Creates a new communications listener	listenerName, listenerAdapter, listenerPort
createAliasConnection	Creates a new Alias Connection	connectionName
createMQConnection	Creates a new MQ Connection	connectionName
createUdpipConnection	Creates a new Udpip Connection	connectionName, address, port
createTcpipHistoryConnection	Creates a new Tcpip History Connection	connectionName, address, port
createTcpipHttpConnection	Creates a new Tcpip Http Connection	connectionName, address, port
createTcpipLengthConnection	Creates a new Tcpip Length Connection	connectionName, address, port
createIndirectConnection	Creates a new Indirect Connection	connectionName, viaQMName
createMQBridgeQueue	Creates a new MQBridge queue	queueName, destinationQMgrName, bridgeName, proxyName, clientConnectionName
createMQBridge	Creates a new MQBridge	bridgeName
triggerTransmission	Initiate the triggering of any pending messages	

Note:

1. See the following table for more information on parameters
2. This operation is provided to allow compatibility with adapters which can not handle array parameters to operations. A similar operation has also been added for some queues and connections.

Operations parameters

Table 18. Queue manager operation parameters

Parameter name	Parameter type	Parameter description
adapter	java.lang.String	Class name for the adapter to use with the message store - optional
address	java.lang.String	IP address for a connection
alias	java.lang.String	Name of the queue manager alias
aliases	[Ljava.lang.String;	Names of the queue manager aliases
bridgeName	java.lang.String	Name of an MQ bridge
clientConnectionName	java.lang.String	Name of MQ client connection associated with an MQ bridge queue
connectionName	java.lang.String	Name of a connection
destinationQMgrName	java.lang.String	Name of the queue manager that owns a given proxy (remote) queue or a bridge queue
forwardToQMgrName	java.lang.String	Name of the queue manager that messages are forwarded to from a Forward queue
getFromQMgrName	java.lang.String	Name of the queue manager that owns a given home server queue
listenerAdapter	java.lang.String	Listener adapter class
listenerName	java.lang.String	Name of a listener
listenerPort	java.lang.String	Port for a listener to listen on
messageStore	java.lang.String	Class name for the message store optional
path	java.lang.String	Path for the queue store optional
port	java.lang.String	IP Port for a connection
proxyName	java.lang.String	Name of MQ queue manager proxy associated with an MQ bridge queue
queueName	java.lang.String	Name of the queue
viaQMName	java.lang.String	Name of a queue manager to connect via (for an indirect connection)

Note:

1. The return type in each case is of type `java.lang.Void`. Hence, return types have not been included in the table.
2. There may seem to be a discrepancy between the input parameters listed for the operations and the input parameters required for the corresponding WMQe operations. This is because the interface design allows the user to input only mandatory parameters at this point. The reason for this is that where the adapter used provides a graphical interface, the inclusion of all optional parameters for each operation would result in a very cluttered interface. Thus, all optional parameters have been omitted in these create operations. Once the resource has been created, they can be specified as updates using `setAttribute()` or `setAttributes()`.
3. Some of these methods may seem unfamiliar to someone who uses the WMQe programmatic interface. In particular the methods

`createStoreQueue()`, and `createForwardQueue()` do not correspond to WMQe standard APIs. The rationale behind these resources is explained in the relevant sections below on Store Queues, Forward Queues and Connections.

Remote queue manager

Attributes

Table 19. Remote queue manager attributes

Attribute name	Attribute description	Attribute type	Read/Write
Accessible	A Boolean value indicating whether the remote queue manager is accessible from the local queue manager or not	java.lang.Boolean	RO
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Aliases	Alternative names for the resource	[Ljava.lang.String;	RW
ChannelAttributeRules	The rule class (or alias) to be associated with the channel attribute	java.lang.String	RW
ChannelTimeout	The time (in milliseconds) after which an outgoing idle channel will be turned off	java.lang.Long	RW
Connections	A list of connections owned by queue manager	[Ljava.lang.String;	RO
MQBridges	A list of Bridges owned by this queue manager	[Ljava.lang.String;	RO
Queues	A list of Queues owned by this queue manager	[Ljava.lang.String;	RO
Rule	The rule class (or alias) to be used by this queue manager	java.lang.String	RW
MaxTransThreads	The maximum number of threads that will be spawned to service the transmission needs of the queue manager	java.lang.Integer	RW
Version	The version of WMQe hosting the queue manager	[Ljava.lang.Short;	RO
CommsListeners	A list of communications listeners owned by this queue manager	[Ljava.lang.String;	RO
BridgeCapable	Indicates whether the queue manager is bridge capable	java.lang.Boolean	RO
MQeClass	The class of the resource	java.lang.String	RO
QMsgStore	The default message store class (or alias) for a queue that determines how messages on that queue are stored	java.lang.String	RW

Table 19. Remote queue manager attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
QAdapter	The default storage adapter class (or alias) for a queue	java.lang.String	RW
QPath	The default path for messages to be stored for queues	java.lang.String	RW

Operations

Table 20. Remote queue manager operations

Operation name	Operation description	Parameter [1]
addAlias [2]	Adds an alias to the queue manager	alias
removeAlias [2]	Removes an alias from the queue manager	alias
addAlias	Adds an array of aliases to the queue manager	aliases
removeAlias	Removes an array of aliases from the queue manager	aliases
createAdminQueue	Creates a new admin queue	queueName, messageStore(optional), adapter(optional), path(optional)
createApplicationQueue	Creates a new application queue	queueName, messageStore(optional), adapter(optional), path(optional)
createHomeServerQueue	Creates a new home server queue.	queueName, getFromQMgrName
createAsyncProxyQueue	Creates a new asynchronous proxy queue	queueName, destinationQMgrName, messageStore(optional), adapter(optional), path(optional)
createSyncProxyQueue	Creates a new synchronous proxy queue	queueName, destinationQMgrName
createStoreQueue	Creates a new store queue	queueName, messageStore(optional), adapter(optional), path(optional)
createForwardQueue	Creates a new forward queue	queueName, forwardToQMgrName, messageStore(optional), adapter(optional), path(optional)
createCommsListener	Creates a new communications listener	listenerName, listenerAdapter, listenerPort
createAliasConnection	Creates a new Alias Connection	connectionName
createMQConnection	Creates a new MQ Connection	connectionName

Table 20. Remote queue manager operations (continued)

Operation name	Operation description	Parameter [1]
createUdpipConnection	Creates a new Udpip Connection	connectionName, address, port
createTcpipHistoryConnection	Creates a new Tcpip History Connection	connectionName, address, port
createTcpipLengthConnection	Creates a new Tcpip Length Connection	connectionName, address, port
createTcpipHttpConnection	Creates a new Tcpip Http Connection	connectionName, address, port
createIndirectConnection	Creates a new Indirect Connection	connectionName, viaQMName
createMQBridgeQueue	Creates a new MQBridge queue	queueName, destinationQMgrName, bridgeName, proxyName, clientConnectionName
createMQBridge	Creates a new MQBridge	bridgeName
refresh	Refresh the queue manager resources from the registry	

Note:

1. See Table 21 for more information on parameters.
2. This operation is provided to allow compatibility adapters which can not handle array parameters to operations. A similar operation has also been added for some queues and connections.

Operations parameters

Table 21. Remote queue manager operation parameters

Parameter name	Parameter type	Parameter description
adapter	java.lang.String	Class name for the adapter to use with the message store - optional
address	java.lang.String	IP address for a connection
alias	java.lang.String	Name of the queue manager alias
aliases	[Ljava.lang.String;	Names of the queue manager aliases
bridgeName	java.lang.String	Name of an MQ bridge
clientConnectionName	java.lang.String	Name of MQ client connection associated with an MQ bridge queue
connectionName	java.lang.String	Name of a connection
destinationQMgrName	java.lang.String	Name of the queue manager that owns a given proxy (remote) queue or a bridge queue
forwardToQMgrName	java.lang.String	Name of the queue manager that messages are forwarded to from a Forward queue
getFromQMgrName	java.lang.String	Name of the queue manager that owns a given home server queue
listenerAdapter	java.lang.String	Listener adapter class
listenerName	java.lang.String	Name of a listener

Table 21. Remote queue manager operation parameters (continued)

Parameter name	Parameter type	Parameter description
listenerPort	java.lang.String	Port for a listener to listen on
messageStore	java.lang.String	Class name for the message store optional
path	java.lang.String	Path for the queue store - optional
port	java.lang.String	IP Port for a connection
proxyName	java.lang.String	Name of MQ queue manager proxy associated with an MQ bridge queue
queueName	java.lang.String	Name of the queue
viaQMName	java.lang.String	Name of a queue manager to connect via (for an indirect connection)

Note:

1. The refresh() operation requires particular comment. When resources are added to or removed from a JMX-enabled queue manager, whether via the JMX interface or from another application, these updates are automatically reflected in the MBeans (that is to say, corresponding MBeans are registered or deregistered). However, when the MBeans corresponding to a queue manager which is remote to the JMX-enabled queue manager (known to it through a direct connection) are updated from another application, these changes are not reflected automatically. In this case, the refresh() operation has to be invoked to update the MBeans in accordance with the current remote queue manager resources.
2. When a direct connection to another Queue Manager is created, a RemoteQueueManager MBean is created in addition to the MBean for the connection definition itself. If the MQeAdminJmx attribute InquireOnConnect is set to true, MBeans for the remote queue manager child resources will be created and registered with the MBeanServer instance at this point. However, if InquireOnConnect is set to false, the child MBeans will not be created. The refresh operation on this MBean will need to be invoked at a later time in order to create the remote queue manager alias MBeans and the MBeans for the child resources when/if required. Note that MBeans for the remote Queue Manager child resources are only created for queue managers connected to the JMX local queue manager by direct connections no child MBeans are created for queue managers known only through MQ, alias or indirect connections.

Admin queue

Attributes

Table 22. Admin queue attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Active	A Boolean value indicating whether the queue is active or not	java.lang.Boolean	RO
Adapter	The adapter class (or alias) to be used by the queue	java.lang.String	RW

Table 22. Admin queue attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
Aliases	Alternative names for the resource	[Ljava.lang.String;	RW
AttributeRule	The attribute class (or alias) associated with the security attributes of the queue	java.lang.String	RW
Authenticator	The authenticator class (or alias) associated with the queue	java.lang.String	RW
Compressor	The compressor class (or alias) associated with the queue	java.lang.String	RW
Cryptor	The cryptor class (or alias) associated with the queue	java.lang.String	RW
CreationDate	The time (in milliseconds since midnight Jan1, 1970 GMT) the queue object was created	java.lang.Long	RO
CurrentDepth	The number of messages currently on the queue	java.lang.Integer	RO
Expiry	The time (in milliseconds) after which messages places on the queue expire.	java.lang.Long	RW
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MaxDepth	The maximum number of messages that may be placed on the queue	java.lang.Integer	RW
MaxMessageSize	The maximum size of a message that can be placed on the queue	java.lang.Integer	RW
MessageStore	The class (or alias) determines how messages on the queue are stored	java.lang.String	RO
MQeClass	The class of the resource	java.lang.String	RO
Path	The path locating the physical storage for the queue	java.lang.String	RO
Priority	The default priority to be associated with messages on the queue	java.lang.Byte	RW
Rule	The rule class (or alias) to be used by the queue	java.lang.String	RW
TargetRegistry	The registry to be used by the authenticator	java.lang.Byte	RW
TimerInterval	The time (in milliseconds) between attempts to get messages	java.lang.Long	RW

Operations

Table 23. Admin queue operations

Operation name	Operation description	Parameter
addAlias	Adds an alias to this queue	alias
removeAlias	Removes an alias from this queue	alias
addAlias	Adds an array of aliases to this queue	aliases
removeAlias	Removes an array of aliases from this queue	aliases
delete	Deletes this queue	

Operations parameters

Table 24. Admin queue operations parameters

Parameter name	Parameter type	Parameter description
alias	java.lang.String	Name of the resource alias
aliases	[Ljava.lang.String;	Names of the resource aliases

Application queue

This queue type is also referred to *local queue* elsewhere in MQe documentation.

Attributes

Table 25. Application queue attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Active	A Boolean value indicating whether the queue is active or not	java.lang.Boolean	RO
Adapter	The adapter class (or alias) to be used by the queue	java.lang.String	RW
Aliases	Alternative names for the resource	[Ljava.lang.String;	RW
AttributeRule	The attribute class (or alias) associated with the security attributes of the queue	java.lang.String	RW
Authenticator	The authenticator class (or alias) associated with the queue	java.lang.String	RW
Compressor	The compressor class (or alias) associated with the queue	java.lang.String	RW
Cryptor	The cryptor class (or alias) associated with the queue	java.lang.String	RW
CreationDate	The time (in milliseconds since midnight Jan1, 1970 GMT) the queue object was created	java.lang.Long	RO
CurrentDepth	The number of messages currently on the queue	java.lang.Integer	RO
Expiry	The time (in milliseconds) after which messages places on the queue expire	java.lang.Long	RW
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MaxDepth	The maximum number of messages that may be placed on the queue	java.lang.Integer	RW
MaxMessageSize	The maximum size of a message that may be placed on the queue	java.lang.Integer	RW
MessageStore	The class (or alias) determines how messages on the queue are stored	java.lang.String	RO

Table 25. Application queue attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
Messages	The message bodies of messages on the queue	[Ljava.lang.String;	RO
MQeClass	The class of the resource	java.lang.String	RO
Path	The path locating the physical storage for the queue	java.lang.String	RO
Priority	The default priority to be associated with messages on the queue	java.lang.Byte	RW
Rule	The rule class (or alias) to be used by the queue	java.lang.String	RW
TargetRegistry	The registry to be used by the authenticator	java.lang.Byte	RW

Operations

Table 26. Application queue operations

Operation name	Operation description	Parameter
addAlias	Adds an alias to this queue	alias
removeAlias	Removes an alias from this queue	alias
addAlias	Adds an array of aliases to this queue	aliases
removeAlias	Removes an array of aliases from this queue	aliases
delete	Deletes this queue	
deleteMessage[1]	Deletes a message from this queue	index
putMessage[1]	Places a message onto the queue	message

Note:

1. See Messaging operations for more details of the messaging operations.

Operations parameters

Table 27. Application queue operations parameters

Parameter name	Parameter type	Parameter description
index	java.lang.Integer	The index of the message to be deleted
message	java.lang.String	The text body of the message to be put
alias	java.lang.String	Name of the resource alias
aliases	[Ljava.lang.String;	Names of the resource aliases

Home Server queue

Attributes

Table 28. Home Server queue attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO

Table 28. Home Server queue attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
Description	An arbitrary string describing the resource	java.lang.String	RW
Active	A Boolean value indicating whether the queue is active or not	java.lang.Boolean	RO
AttributeRule	The attribute class (or alias) associated with the security attributes of the queue	java.lang.String	RW
Authenticator	The authenticator class (or alias) associated with the queue	java.lang.String	RW
Compressor	The compressor class (or alias) associated with the queue	java.lang.String	RW
Cryptor	The cryptor class (or alias) associated with the queue	java.lang.String	RW
CreationDate	The time (in milliseconds since midnight Jan1, 1970 GMT) the queue object was created	java.lang.Long	RO
GetFromQMgr	The name of the queue manager that the home server queue will pull messages from	java.lang.String	RO
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MQeClass	The class of the resource	java.lang.String	RO
Rule	The rule class (or alias) to be used by the queue	java.lang.String	RW
TargetRegistry	The registry to be used by the authenticator	java.lang.Byte	RW
TimerInterval	The time (in milliseconds) between attempts to get messages	java.lang.Long	RW
Transporter	The class (or alias) that flows messages over the channel to the target queue	java.lang.String	RW

Operations

Table 29. Home Server queue operations

Operation name	Operation description	Parameter
delete	Deletes this queue	

Asynchronous Proxy queue

This queue type is also referred to as *asynchronous remote queue* elsewhere in MQe documentation.

Attributes

Table 30. Asynchronous Proxy queue attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO

Table 30. Asynchronous Proxy queue attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
Description	An arbitrary string describing the resource	java.lang.String	RW
Active	A Boolean value indicating whether the queue is active or not	java.lang.Boolean	RO
Adapter	The adapter class (or alias) to be used by the queue	java.lang.String	RW
Aliases	Alternative names for the resource	[Ljava.lang.String;	RW
AttributeRule	The attribute class (or alias) associated with the security attributes of the queue	java.lang.String	RW
Authenticator	The authenticator class (or alias) associated with the queue	java.lang.String	RW
Compressor	The compressor class (or alias) associated with the queue	java.lang.String	RW
Cryptor	The cryptor class (or alias) associated with the queue	java.lang.String	RW
CreationDate	The time (in milliseconds since midnight Jan1, 1970 GMT) the queue object was created	java.lang.Long	RO
CurrentDepth	The number of messages currently on the queue	java.lang.Integer	RO
Expiry	The time (in milliseconds) after which messages places on the queue expire	java.lang.Long	RW
DestinationQMgr	The name of the queue manager to own the physical queue	java.lang.String	RO
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MaxDepth	The maximum number of messages that may be placed on the queue	java.lang.Integer	RW
MaxMessageSize	The maximum size of a message that may be placed on the queue	java.lang.Integer	RW
MessageStore	The class (or alias) determines how messages on the queue are stored	java.lang.String	RO
MQeClass	The class of the resource	java.lang.String	RO
Path	The path locating the physical storage for the queue	java.lang.String	RO
Priority	The default priority to be associated with messages on the queue	java.lang.Byte	RW
Rule	The rule class (or alias) to be used by the queue	java.lang.String	RW
TargetRegistry	The registry to be used by the authenticator	java.lang.Byte	RW
Transporter	The class (or alias) that flows messages over the channel to the target queue	java.lang.String	RW

Operations

Table 31. Asynchronous Proxy queue operations

Operation name	Operation description	Parameter
addAlias	Adds an alias to this queue	alias
removeAlias	Removes an alias from this queue	alias
addAlias	Adds an array of aliases to this queue	aliases
removeAlias	Removes an array of aliases from this queue	aliases
delete	Deletes this queue	
putMessage	Places a message onto the queue	message

Operations parameters

Table 32. Asynchronous Proxy queue operations parameters

Parameter name	Parameter type	Parameter description
message	java.lang.String	The text body of the message to be put
alias	java.lang.String	Name of the resource alias
aliases	[Ljava.lang.String;	Names of the resource aliases

Synchronous Proxy queue

This queue type is also referred to as *synchronous remote queue* elsewhere in MQe documentation.

Attributes

Table 33. Synchronous Proxy queue attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Active	A Boolean value indicating whether the queue is active or not	java.lang.Boolean	RO
Aliases	Alternative names for the resource	[Ljava.lang.String;	RW
AttributeRule	The attribute class (or alias) associated with the security attributes of the queue	java.lang.String	RW
Authenticator	The authenticator class (or alias) associated with the queue	java.lang.String	RW
Compressor	The compressor class (or alias) associated with the queue	java.lang.String	RW
Cryptor	The cryptor class (or alias) associated with the queue	java.lang.String	RW
CreationDate	The time (in milliseconds since midnight Jan1, 1970 GMT) the queue object was created	java.lang.Long	RO
DestinationQMgr	The name of the queue manager to own the physical queue	java.lang.String	RO

Table 33. Synchronous Proxy queue attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MaxMessageSize	The maximum size of a message that may be placed on the queue	java.lang.Integer	RW
MQeClass	The class of the resource	java.lang.String	RO
Rule	The rule class (or alias) to be used by the queue	java.lang.String	RW
TargetRegistry	The registry to be used by the authenticator	java.lang.Byte	RW
Transporter	The class (or alias) that flows messages over the channel to the target queue	java.lang.String	RW

Operations

Table 34. Synchronous Proxy queue operations

Operation name	Operation description	Parameter
addAlias	Adds an alias to this queue	alias
removeAlias	Removes an alias from this queue	alias
addAlias	Adds an array of aliases to this queue	aliases
removeAlias	Removes an array of aliases from this queue	aliases
delete	Deletes this queue	
putMessage	Places a message onto the queue	message

Operations parameters

Table 35. Synchronous Proxy queue operations parameters

Parameter name	Parameter type	Parameter description
message	java.lang.String	The text body of the message to be put
alias	java.lang.String	Name of the resource alias
aliases	[Ljava.lang.String;	Names of the resource aliases

Store queue

These two queue types (`store` and `forward`) require some explanation.

An MQe JMX `store queue` MBean maps onto an MQe queue of type `MQeStoreAndForwardQueue` but with the functionality of that queue somewhat curtailed for ease of use:-

- An `MQeStoreAndForwardQueue` has the ability to store messages for a list of target queue managers (`DestinationQMgrs`) and also has the ability to forward messages to one specified `ForwardToQMgr`.
- However, the MQe JMX implementation has split this dual messaging functionality into two, so that our `store queues` retain the ability to **store** messages for a list of target queue managers, but do not have a `ForwardToQMgr`.
- The **forwarding** functionality of the `MQeStoreAndForwardQueue` is retained in our `forward queue` MBean.

Attributes

Table 36. Store queue attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource.	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Active	A Boolean value indicating whether the queue is active or not	java.lang.Boolean	RO
Adapter	The adapter class (or alias) to be used by the queue	java.lang.String	RW
AttributeRule	The attribute class (or alias) associated with the security attributes of the queue	java.lang.String	RW
Authenticator	The authenticator class (or alias) associated with the queue	java.lang.String	RW
Compressor	The compressor class (or alias) associated with the queue	java.lang.String	RW
Cryptor	The cryptor class (or alias) associated with the queue	java.lang.String	RW
CreationDate	The time (in milliseconds since midnight Jan1, 1970 GMT) the queue object was created	java.lang.Long	RO
CurrentDepth	The number of messages currently on the queue	java.lang.Integer	RO
DestinationQMgrs	The queue manager destinations for which a store (or forward) queue will hold messages	[Ljava.lang.String;	RW
Expiry	The time (in milliseconds) after which messages places on the queue expire	java.lang.Long	RW
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MaxDepth	The maximum number of messages that may be placed on the queue	java.lang.Integer	RW
MaxMessageSize	The maximum size of message that may be placed on the queue	java.lang.Integer	RW
MessageStore	The class (or alias) determines how messages on the queue are stored	java.lang.String	RO
MQeClass	The class of the resource	java.lang.String	RO
Path	The path locating the physical storage for the queue	java.lang.String	RO
Priority	The default priority to be associated with messages on the queue	java.lang.Byte	RW
Rule	The rule class (or alias) to be used by the queue	java.lang.String	RW
TargetRegistry	The registry to be used by the authenticator	java.lang.Byte	RW

Table 36. Store queue attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
Transporter	The class (or alias) that flows messages over the channel to the target queue	java.lang.String	RW

Operations

Table 37. Store queue operations

Operation name	Operation description	Parameter
addDestinationQMgr [1]	Adds a destinationQMgr to the queues DestinationQMgrs List	DestinationQMgr
removeDestinationQMgr [1]	Removes a destinationQMgr from the queues DestinationQMgrs List	DestinationQMgr
addDestinationQMgrs	Adds an array of destinationQMgrs to the queues DestinationQMgrs List	DestinationQMgrs
removeDestinationQMgrs	Removes an array of destinationQMgrs from the queues DestinationQMgrs List	DestinationQMgrs
delete	Deletes this queue	

Note:

1. This operation is provided to allow compatibility with adapters which cannot handle array parameters to operations.

Operations parameters

Table 38. Store queue operations parameters

Parameter name	Parameter type	Parameter Description
DestinationQMgr	java.lang.String	Destination queue manager name to be added or removed
DestinationQMgrs	[Ljava.lang.String;	Destination queue manager names to be added or removed

Forward queue

These two queue types (`store` and `forward`) require some explanation.

An MQe JMX `store queue` MBean maps onto an MQe queue of type `MQeStoreAndForwardQueue` but with the functionality of that queue somewhat curtailed for ease of use:-

- An `MQeStoreAndForwardQueue` has the ability to store messages for a list of target queue managers (`DestinationQMgrs`) and also has the ability to forward messages to one specified `ForwardToQMgr`.
- However, the MQe JMX implementation has split this dual messaging functionality into two, so that our `store queues` retain the ability to **store** messages for a list of target queue managers, but do not have a `ForwardToQMgr`.
- The **forwarding** functionality of the `MQeStoreAndForwardQueue` is retained in our `forward queue` MBean.

Attributes

Table 39. Forward queue attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Active	A Boolean value indicating whether the queue is active or not	java.lang.Boolean	RO
Adapter	The adapter class (or alias) to be used by the queue	java.lang.String	RW
AttributeRule	The attribute class (or alias) associated with the security attributes of the queue	java.lang.String	RW
Authenticator	The authenticator class (or alias) associated with the queue	java.lang.String	RW
Compressor	The compressor class (or alias) associated with the queue	java.lang.String	RW
Cryptor	The cryptor class (or alias) associated with the queue	java.lang.String	RW
CreationDate	The time (in milliseconds since midnight Jan1, 1970 GMT) the queue object was created	java.lang.Long	RO
CurrentDepth	The number of messages currently on the queue	java.lang.Integer	RO
DestinationQMgrs	The queue manager destinations for which a forward (or store) queue will hold messages	[Ljava.lang.String;	RW
Expiry	The time (in milliseconds) after which messages places on the queue expire	java.lang.Long	RW
ForwardToQMgr	The name of the next queue manager that will receive the messages for a forward queue	java.lang.String	RO
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MaxDepth	The maximum number of messages that may be placed on the queue	java.lang.Integer	RW
MaxMessageSize	The maximum size of a message that may be placed on the queue	java.lang.Integer	RW
MessageStore	The class (or alias) determines how messages on the queue are stored	java.lang.String	RO
MQeClass	The class of the resource	java.lang.String	RO
Path	The path locating the physical storage for the queue	java.lang.String	RO
Priority	The default priority to be associated with messages on the queue	java.lang.Byte	RW
Rule	The rule class (or alias) to be used by the queue	java.lang.String	RW

Table 39. Forward queue attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
TargetRegistry	The registry to be used by the authenticator	java.lang.Byte	RW
Transporter	The class (or alias) that flows messages over the channel to the target queue	java.lang.String	RW

Operations

Table 40. Forward queue operations

Operation name	Operation description	Parameter
addDestinationQMgr	Adds a destinationQMgr to the queues DestinationQMgrs List	DestinationQMgr
removeDestinationQMgr	Removes a destinationQMgr from the queues DestinationQMgrs List	DestinationQMgr
addDestinationQMgrs	Adds an array of destinationQMgrs to the queues DestinationQMgrs List	DestinationQMgrs
removeDestinationQMgrs	Removes an array of destinationQMgrs from the queues DestinationQMgrs List	DestinationQMgrs
delete	Deletes this queue	

Operations parameters

Table 41. Forward queue operations parameters

Parameter name	Parameter type	Parameter Description
DestinationQMgr	java.lang.String	Destination queue manager name to be added or removed
DestinationQMgrs	[Ljava.lang.String;	Destination queue manager names to be added or removed

Communications Listener

Attributes

Table 42. Communications Listener attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Running	A Boolean value indicating if the listener is running	java.lang.Boolean	RO
Adapter	The class (or alias) of the communications protocol adapter	java.lang.String	RO
ChannelTimeout	The time (in milliseconds) after which an idle incoming connection will be timed out	java.lang.Long	RW

Table 42. Communications Listener attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
CurrentChannels	The number of channels currently open on the communications listener	java.lang.Integer	RO
MaxChannels	The maximum number of channels allowed for the communications listener	java.lang.Integer	RW
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MQeClass	The class of the resource	java.lang.String	RO
Port	The IP port number used by the communications listener to service incoming connection requests	java.lang.String	RO

Operations

Table 43. Communications Listener operations

Operation name	Operation description	Parameter
stop	Starts this listener	
start	Stops this listener	
delete	Deletes this listener	

MQ/Alias connection

- MQ connections are used to define MQ queue managers. The only parameter needed to create one is the connection definition name.
- Alias connections are used as another way to add aliases to a local queue manager. The only parameter that is needed to create one is the connection definition name.

Both these connections may also be known as no-op connections.

Although there are two separate MQeQueueManagerJmx methods for creating MQ and Alias connections, both types of connection share a domain name: `com.ibm.WMQe_<OwningQMName>_MQConnections:name=<ConnectionName>`.

This is because they are identical in practice.

Attributes

Table 44. MQ/Alias Connection attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Aliases	Alternative names for the resource	[Ljava.lang.String;	RW
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO

Operations

Table 45. MQ/Alias Connection operations

Operation name	Operation description	Parameter
addAlias	Adds an alias to this resource	alias
removeAlias	Removes an alias from this resource	alias
addAlias	Adds an array of aliases to this resource	aliases
removeAlias	Removes an array of aliases from this resource	aliases
delete	Deletes this resource	

Operations parameters

Table 46. MQ/Alias connection operations parameters

Parameter name	Parameter type	Parameter description
alias	java.lang.String	Name of the resource alias
aliases	[Ljava.lang.String;	Names of the resource aliases

Direct connection

To create a direct connection, the parameters *adapter*, *port* and *address* are all valid.

- The *port* and *address* values are required.
- The *adapter* is assigned a default value according to the type of direct connection created.

There are several types of connections which fall under this category and which share the same attributes and operations. These are currently:

- Udpip connection, TcpipLength connection, TcpipHttp connection, TcpipHistory connection.
- An instance of each type of connection and its corresponding MBean is created using a type-specific API in the QueueManager MBean (for example `createUdpipConnection()`).

Once created, the type of connection can be distinguished by the value of the Adapter attribute.

For the sake of convenience, these connection types are grouped together in this section under the heading DirectConnection.

Attributes

Table 47. Direct Connection attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Aliases	Alternative names for the resource	[Ljava.lang.String;	RW
LocalQMgr	The name of the queu manager to own the resource	java.lang.String	RO

Table 47. Direct Connection attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
Adapter	The class (or alias) of the communications protocol adapter	java.lang.String	RW
Address	The numeric or string IP address of the machine hosting the remote queue manager	java.lang.String	RW
Channel	The channel class (or alias) to be used in the connection	java.lang.String	RW
Persist [1]	Whether the adapter should be persistent or not	java.lang.Boolean	RW
Port	The IP port number used by the remote queue manager to service incoming requests	java.lang.String	RW
Servlet	Servlet options	java.lang.String	RW

Note:

1. To avoid confusion about how the attribute `Persist` relates to the options
   - `MQeCommunicationsAdapter.MQe_Adapter_PERSIST`
   - `MQeCommunicationsAdapter.MQe_Adapter_NOPERSIST`

   use the following equivalences:
   - Setting `Persist` to `true` is equivalent to setting `MQeCommunicationsAdapter.MQe_Adapter_PERSIST` to `true`.
   - Setting `Persist` to `false` is equivalent to setting `MQeCommunicationsAdapter.MQe_Adapter_NOPERSIST` to `true`.

Operations

Table 48. Direct Connection operations

Operation name	Operation description	Parameter
addAlias	Adds an alias to this resource	alias
removeAlias	Removes an alias from this resource	alias
addAlias	Adds an array of aliases to this resource	aliases
removeAlias	Removes an array of aliases from this resource	aliases
delete	Deletes this resource	

Operations parameters

Table 49. Direct connection operations parameters

Parameter name	Parameter type	Parameter description
alias	java.lang.String	Name of the resource alias
aliases	[Ljava.lang.String;	Names of the resource aliases

Indirect connection

These are connections that use an intermediate queue manager to get to the final destination queue manger.

Indirect connections require that the `viaQMName` parameter is set to the name of the intermediate queue manager.

The only parameters for creating a connection of this type are the `connectionName` and the `viaQMName`.

Attributes

Table 50. Indirect Connection attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource.	java.lang.String	RO
Description	An arbitrary string describing the resource.	java.lang.String	RW
Aliases	Alternative names for the resource.	[Ljava.lang.String;	RW
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
ViaQMName	The name of the queue manager to be used as the ViaQM	java.lang.String	RW

Operations

Table 51. Indirect Connection operations

Operation name	Operation description	Parameter
addAlias	Adds an alias to this resource	alias
removeAlias	Removes an alias from this resource	alias
addAlias	Adds an array of aliases to this resource	aliases
removeAlias	Removes an array of aliases from this resource	aliases
delete	Deletes this resource	

Operations parameters

Table 52. Indorect connection operations parameters

Parameter name	Parameter type	Parameter description
alias	java.lang.String	Name of the resource alias
aliases	[Ljava.lang.String;	Names of the resource aliases

MQ Bridge queue

Attributes

Table 53. MQ Bridge queue attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
Active	A Boolean value indicating whether the queue is active or not	java.lang.Boolean	RO

Table 53. MQ Bridge queue attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
Aliases	Alternative names for the resource	[Ljava.lang.String;	RW
BridgeName	The bridge object that handles the target MQ queue	java.lang.String	RW
ClientConnection	The name of the client connection associated with the queue	java.lang.String	RW
CreationDate	The time (in milliseconds since midnight Jan1, 1970 GMT) the queue object was created	java.lang.Long	RO
Expiry	The time (in milliseconds) after which messages places on the queue expire	java.lang.Long	RW
DestinationQMgr	The name of the queue manager to own the physical queue	java.lang.String	RO
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MaxIdleTime	The maximum time (in seconds) that the MQ bridge queue can hold onto an idle connection before it is returned to the connection pool	java.lang.Integer	RW
MaxMessageSize	The maximum size of a message that may be placed on the queue	java.lang.Integer	RW
MQQueueManagerProxy	The target MQ QueueManager associated with the queue	java.lang.String	RW
MQRemoteQueueName	The actual queue name of the remote MQ queue	java.lang.String	RW
MQeClass	The class of the resource	java.lang.String	RO
Rule	The rule class (or alias) to be used by the queue	java.lang.String	RW
Transformer	The transformer class (or alias) converting the message from MQe to MQ format	java.lang.String	RW

Operations

Table 54. MQ Bridge queue operations

Operation name	Operation description	Parameter
addAlias	Adds an alias to this queue	alias
removeAlias	Removes an alias from this queue	alias

Table 54. MQ Bridge queue operations (continued)

Operation name	Operation description	Parameter
addAlias	Adds an array of aliases to this resource	aliases
removeAlias	Removes an array of aliases from this resource	aliases
delete	Deletes this queue	
putMessage	Places a message onto the queue	message

Operations parameters

Table 55. MQ Bridge queue operations parameters

Parameter name	Parameter type	Parameter description
message	java.lang.String	The text body of the message to be put
alias	java.lang.String	Name of the resource alias
aliases	[Ljava.lang.String;	Names of the resource aliases

MQ Bridge

A bridge resource is part of a hierarchy which takes the following form:

- an `MQ Bridge` instance can have one or more `MQ QueueManager Proxy` children.
  - these can have `MQ Client Connection` children
    - which can have `MQ Listener` children

Note: Some bridge resources, when their attributes are changed, will only reflect these changes when the resource has been stopped: for example, this is the case with the `SyncQName` attribute of the `MQ Client Connection`. For more information on this subject see:

- Java API Programming Reference
- C API Programming Reference

Attributes

Table 56. MQ Bridge attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
AdministeredObjectClass	The class name (or alias) used to realize the resource	java.lang.String	RO
Children	The list of child objects	[Ljava.lang.String;	RO
DefaultTransformer	The default transformer class (or alias) used for message conversion	java.lang.String	RW
HeartBeatInterval	The heartbeat pulse interval in minutes	java.lang.Integer	RW
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO

Table 56. MQ Bridge attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
RunState	An integer value representing the running state of the resource	java.lang.Integer	RO
StartupRuleClass	The class name (or alias) of the rule used to start the resource	java.lang.String	RW

Operations

Table 57. MQ Bridge operations

Operation name	Operation description	Parameter
start	Starts this MQBridge	affectChildren
stop	Stops this MQBridge	affectChildren
delete	Deletes this MQBridge	affectChildren
createMQQMgrProxy	Creates a new MQ QueueManager proxy	proxyName

Operations parameters

Table 58. MQ Bridge operations parameters

Parameter name	Parameter type	Parameter description
affectChildren	java.lang.Boolean	A Boolean value indicating whether actions on this resource should affect child objects. [1]
proxyName	java.lang.String	Name of the MQ queue manager proxy

Note:

1. Some adaptors may have defaults. These defaults may differ from the WMQe defaults. For example the Sun RI HtmlAdaptorServer defaults all boolean values to true.

MQ Queue Manager Proxy

Attributes

Table 59. MQ Queue Manager Proxy attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
AdministeredObjectClass	The class name (or alias) used to realize the resource	java.lang.String	RO
BridgeName	Identifies the name of the bridge	java.lang.String	RO
Children	The list of child objects	[Ljava.lang.String;	RO
HostName	The IP address of the target MQ queue manager	java.lang.String	RW
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO

Table 59. MQ Queue Manager Proxy attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
RunState	An integer value representing the running state of the resource	java.lang.Integer	RO
StartupRuleClass	The class name (or alias) of the rule used to start the resource	java.lang.String	RW

Operations

Table 60. MQ Queue Manager Proxy operations

Operation name	Operation description	Parameter
start	Starts this MQ Queue Manager Proxy	affectChildren
stop	Stops this MQ Queue Manager Proxy	affectChildren
delete	Deletes this MQ Queue Manager Proxy	affectChildren
createClientConnection	Creates a new MQ Client Connection	clientConnectionName

Operations parameters

Table 61. MQ Queue Manager Proxy operations parameters

Parameter name	Parameter type	Parameter description
affectChildren	java.lang.Boolean	A Boolean value indicating whether actions on this resource should affect child objects
clientConnectionName	java.lang.String	Name of the MQ client connection

MQ Client Connection

Attributes

Table 62. MQ Client Connection attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
AdapterClass	The bridge adapter class (or alias) used to move messages from WMQe to the target MQ queue	java.lang.String	RW
AdministeredObjectClass	The class name (or alias) used to realize the resource	java.lang.String	RO
BridgeName	Identifies the name of the bridge	java.lang.String	RO
CCSID	The CCSID property used by MQ	java.lang.Integer	RW
Children	The list of child objects	[Ljava.lang.String;	RO

Table 62. MQ Client Connection attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MQPassword	The password used with the MQUserID	java.lang.String	RW
MQQMgrProxyName	Identifies the name of the MQ Proxy	java.lang.String	RO
MQUserID	The user ID used by MQ	java.lang.String	RW
MaxConnectionIdleTime	The time (in minutes) after which an idle connection to MQ is discarded and the resources returned to the pool	java.lang.Integer	RW
Port	The IP port number used by the target MQ queue manager	java.lang.String	RW
ReceiveExit	The receive exit specified at the remote end of the MQ client channel	java.lang.String	RW
RunState	An integer value representing the running state of the resource	java.lang.Integer	RO
SecurityExit	The security exit specified at the remote end of the MQ client channel	java.lang.String	RW
SendExit	The send exit specified at the remote end of the MQ client channel	java.lang.String	RW
StartupRuleClass	The class (or alias) of the rule used to start the resource	java.lang.String	RW
SyncQName	The name of the synchronization queue on the MQ queue manager used by the MQBridge	java.lang.String	RW
SyncQPurgeInterval	The time (in minutes) between successive purges of the sync queue	java.lang.Integer	RW
SyncQPurgerRulesClass	The rule class (or alias) used when a message on the sync queue indicates a failure of MQ to confim a message	java.lang.String	RW

Operations

Table 63. MQ Client Connection operations

Operation name	Operation description	Parameter
start	Starts this Client Connection	affectChildren
stop	Stops this Client Connection	affectChildren

Table 63. MQ Client Connection operations (continued)

Operation name	Operation description	Parameter
delete	Deletes this Client Connection	affectChildren
createListener	Creates a new MQ listener	listenerName

Operations parameters

Parameter name	Parameter type	Parameter description
affectChildren	java.lang.Boolean	A Boolean value indicating whether actions on this resource should affect child objects
listenerName	java.lang.String	Name of the MQ listener

MQ Listener

Attributes

Table 64. MQ Listener attributes

Attribute name	Attribute description	Attribute type	Read/Write
Name	The name of the resource	java.lang.String	RO
Description	An arbitrary string describing the resource	java.lang.String	RW
AdministeredObjectClass	The class name (or alias) used to realize the resource	java.lang.String	RO
BridgeName	Identifies the name of the bridge	java.lang.String	RO
ClientConnectionName	Identifies the name of the Client Connection	java.lang.String	RO
DeadLetterQName	The MQ queue used to hold messages that cannot be delivered from MQ to WMQe	java.lang.String	RW
FlowsPerCommit	The number of messages flowed after which the MQ sync queue (if used) is cleared	java.lang.Integer	RW
ListenerStateStoreAdapter	The specification of permanent storage used to hold state information as messages are moved from MQ to WMQe	java.lang.String	RW
LocalQMgr	The name of the queue manager to own the resource	java.lang.String	RO
MQQMgrProxyName	Identifies the name of the MQ Proxy	java.lang.String	RO

Table 64. MQ Listener attributes (continued)

Attribute name	Attribute description	Attribute type	Read/Write
RunState	An integer value representing the running state of the resource	java.lang.Integer	RO
StartupRuleClass	The class (or alias) of the rule used to start the resource	java.lang.String	RW
TransformerClass	The class (or alias) of the actual transformer used by the MQBridge	java.lang.String	RW
UndeliveredMessageRuleClass	The rule class (or alias) determining the action to be taken when a message cannot be delivered from MQ to WMQe	java.lang.String	RW

Operations

Table 65. MQ Listener operations

Operation name	Operation description	Parameter
start	Starts this MQ Listener	
stop	Stops this MQ Listener	
delete	Deletes this MQ Listener	

Glossary

This glossary describes terms used in this book, and words used with other than their everyday meaning. In some cases, a definition might not be the only one applicable to a term, but it gives the particular sense in which the word is used in this book.

If you do not find the term you are looking for, try a softcopy search, or see the hardcopy index, or see the *IBM Dictionary of Computing*, New York:. McGraw-Hill, 1994.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

application programming interface (API)
An application programming interface consists of the functions and variables that programmers are allowed to use in their applications.

asynchronous messaging
A method of communication between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with synchronous messaging.

authenticator
A program that verifies the senders and receivers of messages.

B

bridge
A component that can be added to an MQe queue manager to allow it to communicate with MQ. See MQe queue managers.

C

channel
See *dynamic channel* and *MQI channel*.

channel manager
an MQe object that supports logical multiple concurrent communication pipes between end points.

class
An encapsulated collection of data and methods to operate on the data. A class may be instantiated to produce an object that is an instance of the class.

client
In MQ, a client is a run-time component that allows local user applications to send messages to a server.

compressor
A program that compacts a message to reduce the volume of data to be transmitted.

connection
Links MQe devices and transfers synchronous and asynchronous messages and responses in a bidirectional manner.

© Copyright IBM Corp. 2000, 2004

cryptor
: A program that encrypts a message to provide security during transmission.

D

device platform
: A small computer that is capable of running MQe only as a client, that is, with a device queue manager only.

device queue manager
: See MQe queue managers.

E

encapsulation
: An object oriented programming technique that makes an object's data private or protected and allows programmers to access and manipulate the data only through method calls.

G

gateway
: A computer of any size running an MQe gateway queue manager, which includes the MQ bridge function. See MQe queue managers.

gateway queue manager
: A queue manager with a listener and a bridge. See MQe queue managers.

H

Hypertext Markup Language (HTML)
: A language used to define information that is to be displayed on the World Wide Web.

I

instance
: An object. When a class is instantiated to produce an object, the object is an instance of the class.

interface
: A class that contains only abstract methods and no instance variables. An interface provides a common set of methods that can be implemented by subclasses of a number of different classes.

internet
: A cooperative public network of shared information. Physically, the Internet uses a subset of the total resources of all the currently existing public telecommunication networks. Technically, what distinguishes the Internet as a cooperative public network is its use of a set of protocols called TCP/IP (Transport Control Protocol/Internet Protocol).

J

Java Development Kit (JDK)
: A package of software distributed by Sun Microsystems for Java developers. It includes the Java interpreter, Java classes and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

Java Naming and Directory Service (JNDI)
An API specified in the Java programming language. It provides naming and directory functions to applications written in the Java programming language.

L

Lightweight Directory Access Protocol (LDAP)
A client/server protocol for accessing a directory service.

M

message
In message queuing applications, a communication sent between programs.

message queue
See *queue*.

message queuing
A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

method
The object oriented programming term for a function or procedure.

MQ bridge
A computer with a gateway queue manager that can communicate with MQ. See MQe queue managers.

MQ and MQ family
Refers to **WebSphere MQ**, which includes these products:

- **WebSphere MQ Workflow** simplifies integration across the whole enterprise by automating business processes involving people and applications.
- **WebSphere MQ Integrator** is message-brokering software that provides real-time, intelligent, rules-based message routing, and content transformation and formatting.
- **WebSphere MQ Messaging** provides any-to-any connectivity from desktop to mainframe, through business quality messaging, with over 35 platforms supported.

MQ Messaging
Refers to the following **WebSphere MQ** messaging product groups:

- **Distributed messaging:** MQ for Windows NT and Windows 2000, AIX, iSeries®, HP-UX, Solaris, and other platforms
- **Host messaging:** MQ for z/OS®
- **Pervasive messaging:** MQe

MQe Refers to **WebSphere MQ Everyplace**, the MQ pervasive messaging product group .

MQI channel
Connects an MQ client to a queue manager on a server system and transfers MQI calls and responses in a bidirectional manner.

O

object (1) In Java, an object is an instance of a class. A class models a group of things; an object models a particular member of that group. (2) In MQ, an object is a queue manager, a queue, or a channel.

P

package
: A package in Java is a way of giving a piece of Java code access to a specific set of classes. Java code that is part of a particular package has access to all the classes in the package and to all non-private methods and fields in the classes.

personal digital assistant (PDA)
: A pocket sized personal computer.

private
: A private field is not visible outside its own class.

protected
: A protected field is visible only within its own class, within a subclass, or within packages of which the class is a part.

public
: A public class or interface is visible everywhere. A public method or variable is visible everywhere that its class is visible.

Q

queue
: A queue is an MQ object. Message queueing applications can put messages on, and get messages from, a queue.

queue manager
: A queue manager is a system program that provides message queuing services to applications.

queue queue manager
: This term is used in relation to a remote queue definition. It describes the remote queue manager that owns the local queue that is the target of a remote queue definition. See more at Configuring remote queues - Introduction.

device queue manager
: On **MQe**:- A queue manager with no listener component, and no bridge component. It therefore can only send messages, it cannot receive them.

server queue manager
: On **MQe**:- A queue manager that can have a listener added. With the listener it can receive messages as well as send them.

gateway queue manager
: On **MQe**:- A queue manager that can have a listener and a bridge added. With the listener it can receive messages as well as send them, and with the bridge it can communicate with MQ.

R

registry
: Stores the queue manager configuration information.

S

server

1. An MQe server is a device that has an MQe channel manager configured, and responds to requests for information in a client-server setup.
2. An MQ server is a queue manager that provides message queuing services to client applications running on a remote workstation.

3. More generally, a server is a program that responds to requests for information in the particular two-program information-flow model of client-server.
4. The computer on which a server program runs.

server queue manager
: A queue manager with a listener that can therefore receive messages as well as send them. See MQe queue managers.

server platform
: A computer of any size that is capable of running MQe as a server or client.

servlet
: A Java program which is designed to run only on a Web server.

subclass
: A subclass is a class that extends another. The subclass inherits the public and protected methods and variables of its superclass.

superclass
: A superclass is a class that is extended by some other class. The superclass's public and protected methods and variables are available to the subclass.

synchronous messaging
: A method of communicating between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *asynchronous messaging*.

T

Transmission Control Protocol/Internet Protocol (TCP/IP)
: A set of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

transformer
: A piece of code that performs data or message reformatting.

W

Web See World Wide Web.

Web browser
: A program that formats and displays information that is distributed on the World Wide Web.

World Wide Web (Web)
: The World Wide Web is an Internet service, based on a common set of protocols, which allows a particularly configured server computer to distribute documents across the Internet in a standard way.

Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,

© Copyright IBM Corp. 2000, 2004

Hursley Park,
Winchester,
Hampshire
England
SO21 2JN

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Trademarks

The following terms are trademarks of International Business machines Corporation in the United States, or other countries, or both.

AIX Everyplace IBM iSeries MQSeries WebSphere z/OS zSeries

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Other company, product, and service names may be trademarks or service marks of others.

Printed in USA