MQSeries® Everyplace for
Multiplatforms

Programming Guide

Version 1.1

SC34-5845-01

MQSeries® Everyplace for Multiplatforms

Programming Guide

Version 1.1

SC34-5845-01

Take Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix B. Notices" on page 217

Licence warning

MQSeries Everyplace for Multiplatforms Version 1.1 is a toolkit that enables users to write MQSeries Everyplace applications and to create an environment in which to run them.

The licence conditions under which the toolkit is purchased determine the environment in which it can be used:

*If MQSeries Everyplace for Multiplatforms is purchased for use as a **device** (client) it may **not** be used to create an **MQSeries Everyplace channel manager**, or an **MQSeries Everyplace channel listener.**, or an **MQSeries Everyplace bridge***

*The presence of an **MQSeries Everyplace channel manager**, or an **MQSeries Everyplace channel listener**, or an **MQSeries Everyplace bridge** defines a **gateway** (server) environment, which requires a gateway licence.*

Second Edition (August 2000)

This edition applies to MQSeries Everyplace for Multiplatforms Version 1.1 and to all subsequent releases and modifications until otherwise indicated in new editions.

This document is continually being updated with new and improved information. For the latest edition, please see the MQSeries family library Web page at `http://www.ibm.com/software/ts/mqseries/library/`.

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

Contents

© Copyright IBM Corp. 2000

About this book

This book is a programming guide for the MQSeries Everyplace for Multiplatforms product (generally referred to in this book as MQSeries Everyplace). It contains information on how to use the MQSeries Everyplace class libraries that are described in *MQSeries Everyplace for Multiplatforms, Programming Reference*. It provides guidance to help you to decide which classes to use for common messaging tasks, and in many cases example code is supplied.

The "Chapter 1. Overview" on page 1 provides a brief introduction for those who are unfamiliar with the concepts and components of MQSeries Everyplace. "Chapter 2. Getting Started" on page 9 provides help for setting up your environment, and shows you how to use examples to create applications. The rest of the book contains more detailed information about various aspects of programming with MQSeries Everyplace.

You should use this book in conjunction with the *MQSeries Everyplace for Multiplatforms, Programming Reference* and existing books or manuals on Java® programming.

This document is continually being updated with new and improved information. For the latest edition, please see the MQSeries family library Web page at `http://www.ibm.com/software/mqseries/library/`.

Who should read this book

This book is intended for anyone who wants to write Java based MQSeries Everyplace programs to exchange secure messages between MQSeries Everyplace systems and other members of the MQSeries family of messaging and queueing products.

For information on the availability of development kits for environments other than Java, see the MQSeries Web site at `http://www.ibm.com/software/ts/mqseries/`

Prerequisite knowledge

This book assumes that the reader has a working knowledge of Java and object oriented programming techniques.

An initial understanding of the concepts of secure messaging is an advantage. If you do not have this understanding, you may find it useful to read the following MQSeries books:

- *MQSeries An Introduction to Messaging and Queuing*
- *MQSeries for Windows NT® V5R1 Quick Beginnings*

These books are available in softcopy form from the Book section of the online MQSeries library. This can be reached from the MQSeries Web site, URL address `http://www.ibm.com/software/ts/MQSeries/library/`

© Copyright IBM Corp. 2000

Summary of changes

This section describes changes to this edition of *MQSeries Everyplace for Multiplatforms, Programming Guide*. Changes since the previous edition of the book are marked by vertical lines to the left of the changes.

Changes for this edition (SC34-5845-01)

Some of the information in this book has been restructured to reduce duplication and repetition. Minor errors and omissions have also been corrected.

The following information has been added:

- Details for using MQSeries Everyplace on AIX® and Solaris.
- Readers comment form.

© Copyright IBM Corp. 2000

changes

Chapter 1. Overview

This chapter gives a brief description of MQSeries Everyplace objects and their uses.

MQSeries Everyplace client

The MQSeries Everyplace client is MQSeries Everyplace code that runs on a pervasive or mobile device. Client applications can make use of any application programming interfaces (APIs) or functions available on the device. They are not restricted to just the MQSeries Everyplace programming interfaces.

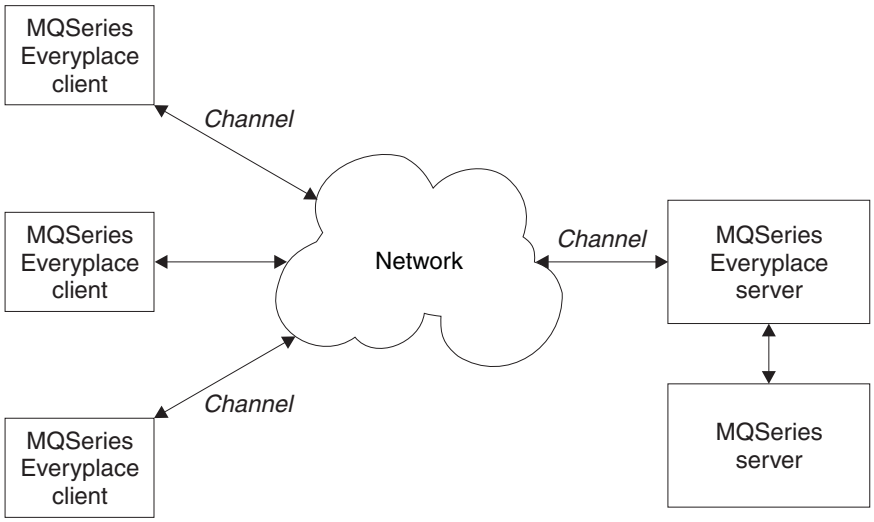

Figure 1. MQSeries Everyplace client

Clients can connect to an MQSeries messaging network by any of the following methods:

- Permanent connection, for example 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

MQSeries Everyplace implements the communications protocols as a set of adapters, one for each of the supported protocols. This enables you to add new protocols very simply, and to tailor the memory footprint on a given client to suit the client's particular requirements.

For more detailed information about MQSeries Everyplace clients, see "Server" on page 42.

© Copyright IBM Corp. 2000

MQSeries Everyplace queue manager

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

- A central point of access to a messaging and queueing network for MQSeries Everyplace applications
- Optional client-side queuing
- Optional administration functions
- Once-only guaranteed delivery of messages
- Full recovery from failure conditions
- Extendable rules-based behavior

The design of the MQSeries Everyplace queue manager is object-oriented. Objects can inherit, and, by the use of rules, you can customize the behavior of the queue manager. The MQSeries Everyplace queue manager can run on a client or as part of an MQSeries Everyplace server.

An MQSeries Everyplace queue manager can controls the various types of queue that are described in "MQSeries Everyplace queues" on page 3. Communication with other queue managers on the MQSeries messaging network can be synchronous or asynchronous.If you want to use synchronous communications, the originator, and the target MQSeries Everyplace queue managers must both be available on the network. Asynchronous communication allows an MQSeries Everyplace application to send messages even when the remote queue manager is offline.

For more detailed information about MQSeries Everyplace queue managers see "Chapter 4. Queue managers, messages, and queues" on page 31

MQSeries Everyplace queues

There are several different types of *queue class* that you can use in an MQSeries Everyplace environment. The types that are available in the MQSeries Everyplace development package are:

- Local
- Remote
- Store-and-forward
- Home-server
- MQSeries-bridge

Queues may have characteristics , such as authentication, compression and encryption. These characteristics are set using attributes, and are used when a message object is stored on a queue.

Local queue

The simplest type of queue is a local queue. These are real queues that are the final destination for all messages. This type of queue is local to, and owned by, a specific queue manager. Applications on the owning queue manager can interact directly with the queue to store messages in safe (excluding hardware failures or loss of the device) and secure way. These queues can be used online or offline, that is connected to a network or not connected to a network.

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.

For more detailed information about local queues, see “Local queue” on page 99.

Remote queue

This type of queue does not reside in the local environment. There is a local queue definition that identifies the real queue and the queue manager that owns it.

You can access remote queues either synchronously or asynchronously. If there is a local definition of the remote queue, the mode of access is based on the definition. In this case, the mode of access may be either synchronous or asynchronous. However, if there is no local definition, *queue discovery* occurs. MQSeries Everyplace retrieves the characteristics (authentication, cryptography, and compression) from the real queue, and forces the mode of access to synchronous.

For more information on remote queues, see “Remote queue” on page 102.

Store-and-forward queue

A store-and-forward queue stores messages until the next queue manager is ready to receive them. (This may not be the owning queue manager.) This type of queue is normally defined on a server, and a client collects its messages when the client connects to the network.

Store-and-forward queues can hold messages for many clients, or there may be one store-and-forward queue per client.

Fore more detailed information about store-and-forward queues, see “Store-and-forward queue” on page 106.

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.

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.

For more detailed information about home-server queues, see "Home-server queue" on page 109.

MQSeries-bridge queue

This type of queue is always defined on an MQSeries Everyplace server and provides a path from the MQSeries Everyplace environment to the MQSeries environment. The MQSeries-bridge queue is a remote queue definition that refers to a queue residing on an MQSeries queue manager.

For more detailed information about the MQSeries-bridge queue, see "MQSeries-bridge queue" on page 110

Administration queue

The administration queue is a specialized queue that understands how to process 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.

MQSeries Everyplace server

The MQSeries Everyplace server is the code that runs on an MQSeries Everyplace gateway and provides the connection mechanism between clients and an MQSeries messaging network. There is no guarantee that the entire network uses a single communications protocol (for example Transmission Control Protocol/Internet Protocol (TCP/IP)). This means that an MQSeries Everyplace server must be able to forward messages from one communications protocol to another. See Figure 2.

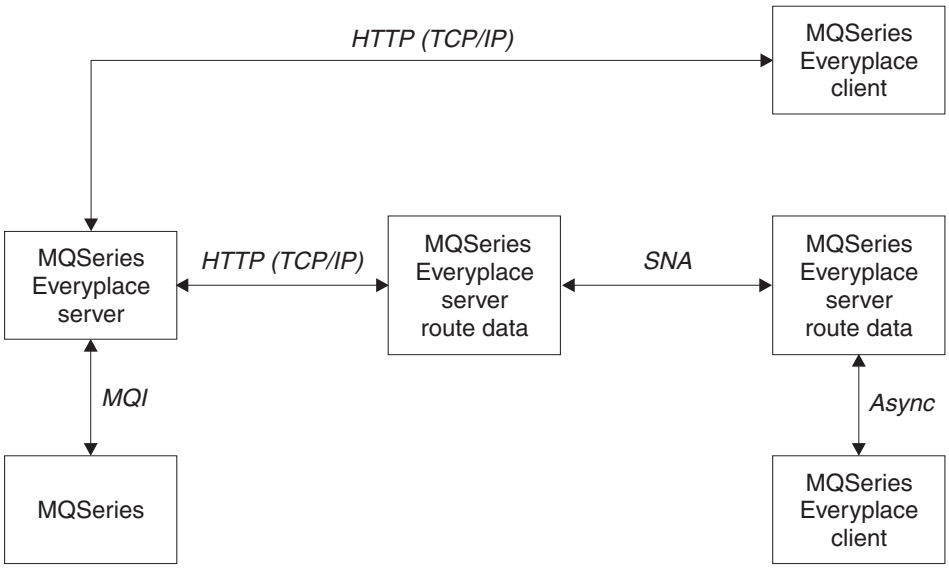

Figure 2. MQSeries Everyplace server

For more detailed information about MQSeries Everyplace servers, see "Server" on page 42.

MQSeries Everyplace bridge to MQSeries

The MQSeries Everyplace server can be an interface to an MQSeries server. The MQSeries-bridge handles the transfer of messages between the two systems. “Configuring the MQSeries-bridge” on page 119 provides a detailed description of this interface.

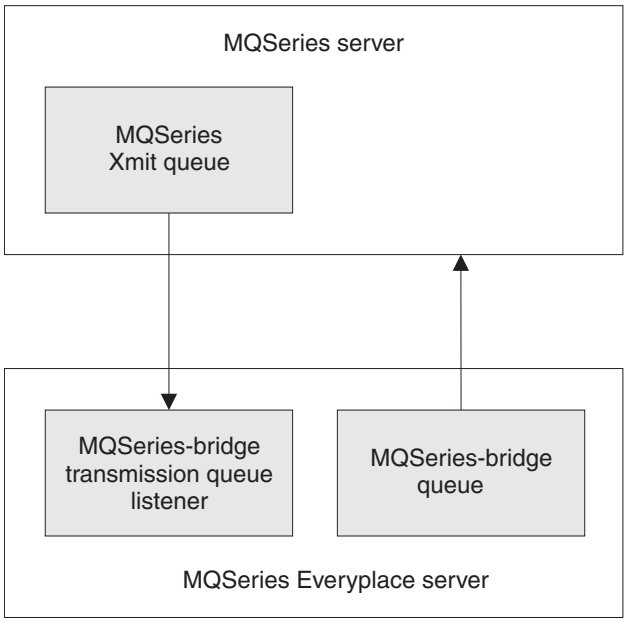

Figure 3. MQSeries Everyplace interface to MQSeries

MQSeries Everyplace channels

MQSeries Everyplace supports a method of establishing connections between queue managers, that is termed an MQSeries Everyplace *channel*. A channel is a logical connection between the two parties, and is established for the purposes of sending or receiving data. Channels can have various attributes or characteristics, such as authentication, cryptography, compression, or the transmission protocol to use. Different channels can use different characteristics. Each channel can have its own value set for each of the following attributes:

Authenticator
Either *null* or an authenticator object that can perform user or channel authentication.

Cryptor
Either *null* or a cryptor object that can perform encryption and decryption

Compressor
Either 'null', or a compressor object that can perform data compression and decompression.

Destination
The target for this channel, for example `server.xyz.com`

You typically use the authenticator only when setting up the channel. Compressors and cryptors are normally used on all flows.

For more detailed information about channels see "Connections" on page 93, and for more information about authenticators, compressors and cryptors, see "Chapter 8. Security" on page 157.

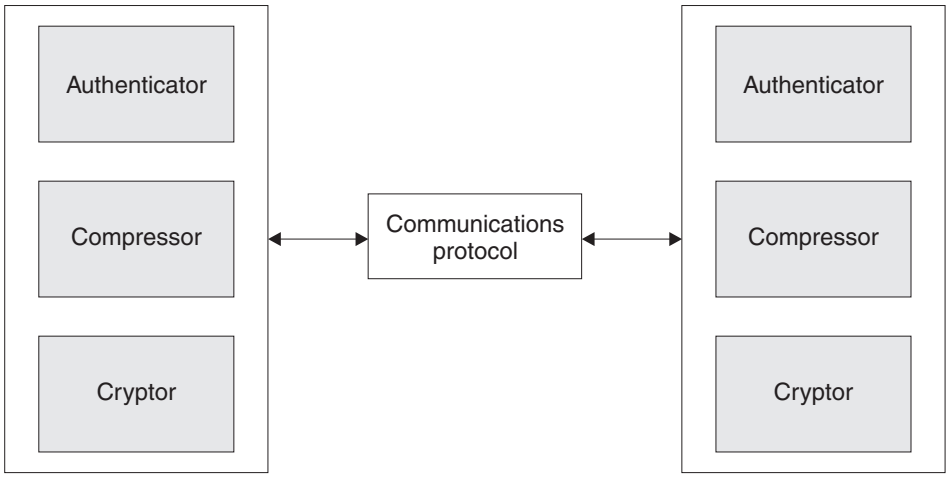

Figure 4. MQSeries Everyplace channel

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. MQSeries Everyplace provides facilities for this, but it is the responsibility of the application programmer to implement it.

Security

MQSeries Everyplace includes an integrated set of security features that provide protection for message data, when it is held locally, and when it is being transferred. There are three different categories of security:

Local security
Local security provides protection for MQSeries Everyplace messages while they are held by a local queue manager.

Queue-based security
Queue-based security automatically protects MQSeries Everyplace message data between an initiating queue manager and a target queue, so long as the target queue is defined with an attribute. This protection is independent of whether the target queue is owned by a local or a remote queue manager.

Message-level security
Message-level security provides protection for message data between an initiating and receiving MQSeries Everyplace application.

MQSeries Everyplace local and message-level security are made available for applications to use. MQSeries Everyplace queue-based security is an internal service.

The high security version of MQSeries Everyplace also provides a mini-certificate server for enhanced security.

See “Chapter 8. Security” on page 157 for detailed information about MQSeries Everyplace security features.

Chapter 2. Getting Started

This section introduces Version 1.1 of the MQSeries Everyplace Development Kit. The Development Kit is a development environment for writing messaging and queuing applications. based on Java 1.1.

Note: For information on the availability of development kits for environments other than Java, see the MQSeries Web site at `http://www.ibm.com/software/ts/mqseries/`

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

Base MQSeries Everyplace classes
A set of Java classes that provide all the necessary function to build messaging and queuing applications.

Examples
A set of Java source code and classes that demonstrate how to use many features of MQSeries Everyplace.

Development Environment

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

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

  Windows

  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 MQSeries–bridge, the MQSeries Java client must be installed and made available to the JDK. For details on setting up the environment for the MQSeries Java client, see *MQSeries Using Java*.

You can use many different Java development environments and Java runtime environments with MQSeries Everyplace. The system configuration for both development and runtime is dependent on the environment used. MQSeries Everyplace 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 MQSeries Everyplace 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.

© Copyright IBM Corp. 2000

- Set the *JavaCmd* environment variable to the command used to run Java applications.
- If the MQSeries Java Client is installed, set the *MQDIR* environment variable to the base directory of the MQSeries Java client.

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

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

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

Deploying applications

When deploying MQSeries Everyplace 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 resource. MQSeries Everyplace provides the following examples of how the MQSeries Everyplace classes can be packaged into jar files. These examples are in the <MQeInstallDir>\Java\Jars directory of a standard MQSeries Everyplace installation.

MQeDevice.jar
A full set of the base classes that can be used on a device

MQeGateway.jar
A full set of the base classes that can be used on a gateway

MQeMQBridge.jar
The classes that can be used to extend the MQeGateway.jar to build a gateway that interoperates with MQSeries

MQeHighSecurity.jar *
A set of classes that can be used to extend both the MQeGateway.jar and MQeDevice.jar to provide enhanced security

MQeMiniCertificateServer.jar *
A self contained jar file providing all the classes required to run the mini-certificate server

MQeExamples.jar
A packaging of all the MQSeries Everyplace examples into one jar file

* These jar files are only included in the high security edition of MQSeries Everyplace.

To run MQSeries Everyplace applications, you must set up the Java runtime environment to include the required MQSeries Everyplace and application classes. Using a standard Java runtime environment (JRE), you must set the CLASSPATH to include any required `jar` files.

Example statements are:

Windows

```
Set CLASSPATH=<MQeInstallDir>\Jars\MQeDevice.jar;%CLASSPATH%
```

UNIX

```
CLASSPATH=<MQeInstallDir>/Java/Jars/MQeDevice.jar:$CLASSPATH
export CLASSPATH
```

Post install test

Once you have installed MQSeries Everyplace you can use the following procedures to run a set of examples that determine whether the installation of the development kit was successful.

- Ensure that the Java environment is set up as described in "Development Environment" on page 9. When running any of the Windows batch files described in this section, the first parameter of each is the name of the Java development kit to use, if you do not specify a name, the default is IBM.

  Note: The UNIX shell scripts do not have a corresponding parameter.

- **Windows**
  Change to the <MQeInstallDir>\Java directory.

  UNIX Change to the <MQeInstallDir>/Java/demo/UNIX directory.

- Create a queue manager as follows:

  Windows
  Run the batch file

  ```
  CreateExampleQM.bat <JDK>
  ```

  UNIX Run the shell script

  ```
  CreateExampleQM
  ```

  to create an example queue manager called `ExampleQM`.

  Part of the creation process sets up directories to hold queue manager configuration information and queues. The example uses a directory called ExampleQM that is relative to the current directory. Within this directory are two other directories:

  – Registry - holds files that contain queue manager configuration data.
  – Queues - for each queue there is a subdirectory to hold the queue's messages. (The directory is not created until the queue is activated.)

- Run a simple application as follows:

  Once you have created a queue manager you can start it and use it in applications. You can use the batch file ExamplesMQeClientTest.bat or the shell script ExamplesMQeClientTest to run some of the simple application examples.

The batch file runs examples.application.Example1 by default. This example puts a test message to queue manager ExampleQM and then gets the message from the same queue manager. If the two messages match, the application ran successfully.

There are a set of applications in the examples.application package that demonstrate different features of MQSeries Everyplace. You can run these examples as follows:

Windows
Pass parameters to the batch files:

```
ExamplesMQeClientTest <JDK> <ExampleNo>
```

UNIX Pass parameters to the shell scripts:

```
ExamplesMQeClientTest <ExampleNo>
```

where *ExampleNo* is the suffix of the example. This can range from 1 to 6.

- Delete a Queue manager.

  When a queue manager is no longer required you can delete it. To delete the example queue manager ExampleQM:

  Windows
  Run the batch file

  ```
  DeleteExampleQM.bat <JDK>
  ```

  UNIX Run the shell script

  ```
  DeleteExampleQM
  ```

  .

  Once you have deleted a queue manager you cannot start it.

  Notes:

  1. Deleting a queue manager does not delete any messages that are still on the queue, or configuration data that was not part of the base queue manger creation. Hence, if the queue manager is recreated with the same creation parameters, the remaining messages are available to the recreated queue manager.
  2. The examples use relative directories for ease of set up. You are strongly recommended to use absolute directories for anything other than base development and demonstration. If the current directory is changed, and you are using relative directories, the queue manager can no longer locate its configuration information and queues.

Examples

The examples previously described form a small part of the set of examples provided with MQSeries Everyplace. Each example demonstrates how to use or extend a feature of MQSeries Everyplace. Most are described in the relevant sections of this Guide. They are all listed and briefly described in the following sections

examples.application package

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 "Post install test" on page 11).

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.

Example4
Use the **WaitForMessage** method to get a message if it arrives within a specified interval.

Example5
Lock messages then get, unlock, and delete them.

Example6
Simple put and get of a message using assured message delivery.

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 (see "Development Environment" on page 9 for details). 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 "Chapter 4. Queue managers, messages, and queues" on page 31.

examples.administration.simple package

This package contains a set of examples that show how to use some of the administrative features of MQSeries Everyplace 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

For details of MQSeries Everyplace administration functions, see "Chapter 6. Administering messaging resources" on page 83.

examples.administration.console package

This package contains a set of classes that implement a simple graphical user interface (GUI) for managing MQSeries Everyplace resources.

Admin
Front end to the example administration GUI.

Additionally there is a suite of classes that provides the graphical user interface for each MQSeries Everyplace 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`

See "Chapter 6. Administering messaging resources" on page 83 for more details information about using the MQSeries Everyplace administration functions.

examples.attributes package

This package contains a set of classes that show how to write additional components to extend MQSeries Everyplace security.

NTAuthenticator
An authenticator that authenticates a user to the Windows NT security

database. 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

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` pre-processor 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)).

TableCryptor
A very simple cryptor

See "Chapter 8. Security" on page 157 for more detailed information about the MQSeries Everyplace security features.

examples.awt package

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 MQSeries Everyplace 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" on page 42 for more details about running a queue manager in a server environment.

AwtMQeTrace
A graphical front end to examples.trace.MQeTrace.

See "Chapter 9. Tracing in MQSeries Everyplace" on page 197 for more information about the MQSeries Everyplace trace facility.

Classes AwtDialog, AwtEvent, AwtFormat, AwtFrame, AwtLayout, and AwtOutputStream provide a toolkit for building small footprint awt-based graphical applications. These classes are used by many of the graphical MQSeries Everyplace examples.

examples.eventlog package

This package contains some examples that demonstrate how to log events to different facilities.

LogToDiskFile
Write events to a disk file

examples

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.

examples.install package

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 to 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.

For more details about creating and deleting queue managers, see "Chapter 4. Queue managers, messages, and queues" on page 31.

examples.nativecode package

Several of the examples require access to operating system facilities on Windows NT, or UNIX (AIX and Solaris). MQSeries Everyplace 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`.

examples.queuemanager package

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:

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 gateway. 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

For more details about running queue managers in different environments see "Starting queue managers" on page 36. For details on queue managers that provide an environment for secure queues and messaging (MQePrivateClient and MQePrivateServer), see "Chapter 8. Security" on page 157.

examples.rules package

You can control and extend the base MQSeries Everyplace functionality using rules. Some components of MQSeries Everyplace allow rules classes to be applied to them. 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 "Chapter 5. Rules" on page 73 for more details.

AttributeRule
Example attribute rule that controls the use of attributes.

examples.security package

This package contains an example that modifies MQSeries Everyplace security.

MQeSecurity
An example extension to the Java security manager that controls whether permission is granted to use certain features of MQSeries Everyplace.

examples.trace package

This package contains an example trace handler that can be used for debugging an application during development, and for tracing a completed application.

examples

MQeTrace
The base MQSeries Everyplace 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 MQSeries Everyplace

examples.mqbridge package

This package contains a set of classes that show how to use and extend the MQSeries-bridge. Some of the examples extend other MQSeries Everyplace examples.

See "Chapter 7. MQSeries-bridge" on page 119 for more details about the MQSeries-bridge.

Chapter 3. MQeFields

MQeFields is the fundamental class used to hold data items for sending, receiving, or manipulating MQSeries Everyplace messages. An MQeFields object is constructed as follows:

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

There are various **put** and **get** methods within the MQeFields object for storing and retrieving items. Items are held in a name, type and value form.

The name must conform to the following rules:

- It must be at least 1 character long.
- It must conform to the ASCII character set (characters with values 20 < value < 128).
- It must *not* include any of the characters { } [ ] # ( ) : ; , ' " =
- It must be unique within the MQeFields object

The MQeFields object name is used to retrieve and update values. It is good practice to keep names short, because the names are included with the data when the MQeFields object is dumped.

The following examples shows how to store values in an MQeFields object:

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

The following examples shows how to retrieve values from an MQeFields object:

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

Methods are provided for storing and retrieving the value types shown in Table 1

Table 1. Store and retrieve methods

Value type	Store method	Retrieve method
byte	putByte	getByte
int	putInt	getInt
short	putShort	getShort
long	putLong	getLong
floating point	putFloat	getFloat
	putDouble	getDouble
boolean	putBoolean	getBoolean
string	putAscii	getAscii
	putUnicode	getUnicode

Arrays of values may be held within a fields object. There are two forms for holding arrays:

© Copyright IBM Corp. 2000

- Fixed length arrays are handled using the **putArrayOf***type* and **getArrayOf***type* methods. *type* can be Byte, Short, Int, Long, Float, or Double.
- Variable length arrays are handled using the **put***type***Array**and**get***type***Array**. *type* can be Byte, Short, Int, Long, Float, or Double.

  Using this form, each element is stored as a series of single items. *:nn* is appended to the name of the item, where *nn* is the element number of the item within the array, starting at 0. A separate item contains the array length. This array length is an integer value and is handled using **putArrayLength** and **getArrayLength**.

An MQeFields object may be imbedded within another MQeFields object by using the **putFields** and **getFields** methods.

MQeMsgObject, or a descendant of this class, is used for normal MQSeries Everyplace messages. MQeMsgObject is a descendant of the MQeFields class, and hence has access to all the MQeFields methods. See "Messages" on page 56for more information on MQeMsgObject.

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

binary This is the form normally used to send an MQeFields or MQeMsgObject object through the network. The method used to convert the data to binary is **dump**. This method returns a binary byte array containing an encoded form of the contents of the object (Note: this is not Java serialization). The **dump** method has an optional boolean parameter that specifies if the dumped data is to be XOR'd with a previous copy of the object data. This is an attempt to increase the number of bytes in the output array that are "0x00" to help the compressor make the data stream smaller when it is sent over the network. This parameter is only useful when the application intends to write the byte array out to some other physical media.

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

There are various restrictions placed on the string form and it may not always be possible to restore the MQeFields object using the string. The string form uses the **dumpToString** method of the MQeFields object. 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 imbedded MQeFields objects to a string, but there is no guarantee that the imbedded MQeFields data can be restored using the **restoreFromString** method.

A powerful feature of MQeFields is the ability to read an ini file as in the following example:

```
[Section1]
Keyword1=value1
Keyword2=value2
[Section2]
Keyword1=value
...
```

This data can be read and parsed into an MQeFields object as shown in the following example:

```
File diskFile = new File( fileName );              /*access the file*/
  byte data[] = new byte[(int) diskFile.length()];       /*file size*/
  FileInputStream inputFile = new FileInputStream( diskFile );
  inputFile.read( data );                            /*read the file*/
  inputFile.close( );                              /*finish with file*/

  MQeFields fields = new MQeFields( );          /*new Fields Object*/
  fields.restoreFromString( "\r\n",            /*end of line string*/
"[#0]",                                          /*section pattern*/
"#1=#2",                                          /*keyword pattern*/
byteToAscii( data ) );
```

The following example shows a variation of the code that allows different data types to be restored:

```
[Section1]
(ascii)Keyword1=value1
(int)Keyword2=1234
[Section2]
(boolean)Keyword1=true
...
```

```
[
File diskFile = new File( fileName );               /*access the file*/
  byte data[] = new byte[(int) diskFile.length()];     /*size of file*/
  FileInputStream inputFile = new FileInputStream( diskFile );
  inputFile.read( data );                             /*read the file*/
  inputFile.close( );                               /*finish with file*/

  MQeFields fields = new MQeFields( );            /*new Fields Object*/
  fields.restoreFromString( "\r\n",              /*end of line string*/
"[#0]",                                            /*section pattern*/
"(#0)#1=#2",                                        /*keyword pattern*/
byteToAscii( data ) );
```

Note: The **dumpToString** method does not dump imbedded MQeFields objects in a format that can be restored using the preceding technique.

The use of an ini files is optional, although it is recommended. Utilities for handling ini files are found in the examples supplied with MQSeries Everyplace, but you may prefer to write your own, or to use MQeFields objects directly. If you use ini files, they must be restored into the MQeFields objects in order to create and manage queue managers. Examples are provided to help with this process.

Creating an MQeFields-based ini file editor

This example creates an ini file editor using the example components in the MQSeries Everyplace examples.awt directory. It is not meant to include all forms of MQeFields but is intended as an example, or a starting point, for a more powerful editor.

MQefields-based ini editor

The example treats each section as a separate imbedded MQeFields object. The base class creates a window with a choice box listing all the sections found within the ini file.

The example makes use of the classes in the examples.awt directory. These classes provide a simple way of creating and manipulating basic frames and dialogs.

The application extends `examples.awt.AwtFrame`to create a frame with a menu.

```
public class Editor extends examples.awt.AwtFrame
  /*-----------------------------------------------------------------*/
  public editor( String args[] ) throws Exception
    {
  /* Assign the title to the frame and initializes the ancestor.*/
    super( "Editor - " );
  /*Assign a menu bar to the frame and define the items that appear on the bar*/
    format( Menu,  new String[][][] {
                          { { "File"      },
                            {  " ", "Open"  },       /* Index   0   */
                            {  " ", "Save"  },       /* Index   1   */
                            {  "-" },
                            {  " ", "Exit"  } },     /* Index   2   */
                            { { "Help"    },
                            {  " ", "Trace" } } } ); /* Index   3   */
    visible( true );
    }
```

The **format** method call has two parameters. The first in this example is *Menu* and the second is a *String array* object.

The *String array* must be an array of array of array as follows:

- The first dimension defines the number of rows.
- The second dimension defines the number of columns
- The third dimension defines the components of the menu. These are defined as follows:

  ```
  new String[][][] { type, Data {, Data, { ... } } }
  ```

  where:

  type is the component type. This can be any of the following:

″ ″	normal menu item
″**C**″	CheckItem - unchecked. The modifier ″!″ denotes checked.
″**-**″	separator

  Anything else is treated as a label.

  Data the text to be used by the component.

  Each item on the menu that can cause an action event has an index number based on its position in the array in the preceding code fragment the comments show the index number

The first parameter of the **format** method can have the values of "North", "South", "East", "West", and "Center". These correspond to the position of a panel within the frame. In these cases, the *string array* object has the following syntax:

```
new String[][][] { type, Data {, Data, { ... } } }
```

where:

type is the component type. This can have the following values:

- "**A**" Text area, with the following possible modifiers:
  - "**P**" Protected - can not be edited
  - "**K**" Give Key released action events
- "**B**" Button
- "**C**" Checkbox - unchecked. The modifier "!" denotes checked.
- "**D**" Choice (drop down list)
- "**L**" Label
- "**S**" Selection list (list box)
- "**T**" TextField, with the following possible modifiers:
  - "**K**" give Key released action events
  - "**P**" protected - can not be edited
  - "*" masked input

Any thing else is treated as a label

Data the text to be used by the component

For more information on how these methods work, see the `AwtDialog`, `AwtFormat`, and `AwtFrame` examples in the examples.awt directory.

Using the examples.awt components to create the editor, the following code defines three work variables and the constructor that creates a window with a menu and a single Choice box .

```
public class Editor extends examples.awt.AwtFrame
  {
  protected Choice    choiceBox   = null;
  protected MQeFields fields      = null;
  protected String    currentFile = "";

  /*--------------------------------------------------------------------*/
  public editor( String args[] ) throws Exception
    {
    super( "Editor - " );
    format( Menu,  new String[][][] {
                         { { "File"      },
                           {  " ", "Open"  },       /* Index   0   */
                           {  " ", "Save"  },       /* Index   1   */
                           {  "-" },
                           {  " ", "Exit"  } },     /* Index   2   */
                           { { "Help"    },
                           {  " ", "Trace" } } } ); /* Index   3   */
    format( North, new String[][][] {
                         { { "D", "< -- No File Loaded -- >" } } } );
    choiceBox = (Choice) getObject( North, 0 );
    visible( true );
    }

  /*--------------------------------------------------------------------*/
  public static void main( String[] args )
    {
    try
      {
      new  Editor( args );
      }
```

```
    catch ( Exception e )
      {
      e.printStackTrace();
      }
    }
```

The next piece of code handles the events caused by the user interacting with the menu or the choice box.

The menu actions are:

Open has an Action index of "0". This is used in a switch statement and calls a **Load** method to read a disk file

Save. · has an Action index of "1". This is used in a switch statement and calls a **Save** method to write a disk file

Exit has an Action index of "2". This is used in a switch statement and exits the program

Trace has an Action index of "3". This is used in a switch statement and calls the Examples.Awt.AwtMQeTrace class

The choice box is the only component placed in the North, and therefore it has an index of "0". Selecting an item in this list box activates the class used to display the contents of an MQeFields object.

```
public void action( Object e, int where, int index,
  String choice, boolean state )
    {
    try
      {
      switch ( where )
        {
        /* process Menu actions                                        */
        case Menu:
          switch ( index )
            {
            case  0: load( );          break;
            case  1: save( );          break;
            case  2: System.exit( 0 ); break;
            case  3: new examples.awt.AwtMQeTrace( "Edit Trace", null );
    break;
            }
          break;
        /* process North events                                        */
        case North:
          switch ( index )
            {
            case 0:
              String item = choiceBox.getSelectedItem();
              new EditorFieldsDisplay( "Editor - ["  + Item +  "]",
                                       fields.getFields( Item ) );
              break;
            }
          break;
        }
      }
    /* exception occured - show error in a modal dialog window       */
    catch ( Exception ex )
      {
      ex.printStackTrace();
      new examples.awt.AwtDialog( this,
   "Exception",
   examples.awt.AwtDialog.Show_OK,
```

```
   new String[][][] {
  { { "TP", ex.toString() }  } } );
      }
    }
```

The next piece of code processes the Save Menu request

This code creates a common file dialog and displays it to allow the output file name to be specified. Once the file path and file name are set, each imbedded MQeFields object is dumped to a String variable, creating a single String of the whole MQeFields object. This String is written to disk and the output file is closed.

```
protected void save( ) throws Exception
    {
    if ( fields == null )   throw new Exception( "No Fields object" );
    FileDialog fd  = new FileDialog( this, "", FileDialog.SAVE );
    fd.setFile( CurrentFile );
    fd.show( );
    if ( (fd.getDirectory() != null) && (fd.getFile() != null) )
      {
      currentFile   = fd.getDirectory() + fd.getFile();
      File diskFile = new File( currentFile );
      /* look for imbedded fields objects   */
      String buffer = "";                /*for imbedded_fields objects*/
      String base   = "";                  /*non-imbedded_fields items*/
      Enumeration keys = fields.fields();              /*get the names*/
      while ( keys.hasMoreElements() )
        {
        String key = (String) keys.nextElement();
        if  ( fields.dataType( key )  == MQeField.TypeFields )
          buffer = buffer + "[" + key + "]\r\n" +
                   fields.getFields( key ).dumpToString(
   "(#0)#1=#2\r\n" ) + "\r\n";
        else                                       /*... no, normal item*/
          base   = base   + fields.dumpToString( "(#0)#1=#2\r\n", key );
        }
      buffer = base + buffer;
      FileOutputStream outputFile = new FileOutputStream( diskFile );
      outputFile.write( MQe.asciiToByte( buffer ) );
      outputFile.close( );
      }
    }
```

So far we have:

- Defined the controlling application window
- Defined the process for loading and saving a disk file
- Defined a mechanism for activating EditorFieldsDisplay class once a specific section has been selected

The EditorFieldsDisplay class is where the actual editing is done. This class creates the editor screen. The constructor of the class sets up:

- The menu (in this case is just Exit, in the North)
- A choice box containing any names of imbedded MQeFields objects
- A list box to hold the dumped items, in the center

The next piece of code positions the sub-window on the screen.

```
public class EditorFieldsDisplay extends AwtFrame
  {
  protected MQeFields fields     = null;        /* fields object       */
  protected Choice    choiceBox  = null;        /* Fields Choice       */
  protected List      listBox    = null;        /* listbox object      */
  protected String    newItem    =
```

```
"  <<<< Double click here to add new item >>>>";

/* constructor                                                          */
public EditorFieldsDisplay( String    thisTitle,
                            MQeFields theseFields ) throws Exception
  {
  super( thisTitle );
  fields = theseFields;
  format( Menu,   new String[][][] {
                            { { "Exit"    },
                              {  " ", "Exit"   } } } );/* Index  0   */
  format( North,  new String[][][] {
                            { { "D",  "<none>" } } } );/* Index  0   */
  format( Center, new String[][][] {
                            { { "S",  "" } } } );      /* Index  0   */

  choiceBox = (Choice) getObject( North,  0 );
  listBox   = (List)   getObject( Center, 0 );
  listBox.setFont( new Font( "Courier", 1, 12) );
  visible( true );
  /* re-size/re-position the edit window                               */
  Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
  setSize    ( screenSize.width / 3, screenSize.height / 3 );
  setLocation( screenSize.width / 3, screenSize.height / 3 );
  /* initialise the various component contents                         */
  showFields( ); /* show fields contents  */
  }
```

The **showFields** method call is a call to a common routine that refreshes the data in the list box. This is a list of the items that are held within the MQeFields object.

```
protected void showFields( ) throws Exception
    {
    listBox.removeAll();              /* clear all entries           */
    if ( fields != null )             /* fields object ?             */
      {                               /* ... yes                     */
      Enumeration keys = fields.Fields();       /* get the key names   */
      choiceBox.removeAll();
      while ( keys.hasMoreElements() )
        {
        String key = (String) keys.nextElement();
        if  ( fields.dataType( key )  == MQeField.TypeFields )
          choiceBox.add( key );       /* ... yes, add name           */
        else
          listBox.add( format( fields.dumpToString( "#1\t(#0)\t = #2",
         Key), 10 ) );
        }
      listBox.add( newItem );         /* add information line        */
      }
    }
```

The `ListBox.add( format( fields.dumpToString( "#1\t(#0)\t = #2",` record in the preceding code, dumps the MQeFields data with tab ("\t"), carriage returns ("\r") and line feed ("\n") characters. These need to be formatted before they are displayed in the list box.

The following piece of code shows a formatter.

```
public static String format( String data, int tabSize )
    {
    int l = 0;                                    /*start line number*/
    char c[] = new char[data.length()];                 /*work array*/
    data.getChars( 0, data.length(), c, 0 );      /*convert to chars*/
    StringBuffer result = new StringBuffer( 512 );
    for ( int i = 0; i < c.length; i = i + 1 )
      switch ( c[i] )
        {
```

```
      case '\r': /* ignore */
      case '\n': /* new line */
         l = 0; /* set space count */
         result.append( c[i] );                  /*append to string*/
         break;
      case '\t':                                    /*tab character*/
        int m = l;                                 /*current position*/
        for ( l = m; l < tabSize + 1; l = l + 1 )        /*fill tab*/
          result.append( ' ' );                            /*pad*/
        l = 0;                                           /*reset*/
        break;
      default:                                        /*all others*/
        result.append( c[i] );                    /*append to string*/
        l = (l + 1) % tabSize;                 /*increase the length*/
        break;
      }
    return( result.toString() );
    }
```

The following code handles the events caused by the user interacting with the menu, the choice box or the list box.

Exit is the only menu item, and this is handled by disposing of the edit window. The choice box handles any imbedded MQeFields items. To edit an Individual item, the user must select the item within the list box.

```
  public void action( Object e, int where, int index,
   String choice, boolean state )
    {
    try
      {
      switch ( where )
        {
        /* process Menu events                                        */
        case Menu:
          switch ( index )
            {
            case  0:  dispose( );       break;
            }
          break;
        /* process North events                                       */
        case North:
          switch ( index )
            {
            case  0:         break;
            }
          break;
        /* process Center events                                     */
        case Center:
          switch ( index )
            {
            case 0:
              int i = listBox.getSelectedIndex();
              if  ( i > -1 )                    /* anything selected ? */
                {
                String editName  = listBox.getItem( i );
                if ( editName.equals( newItem ) )    /* add new item ? */
                  editItem( "", "ascii", "" );  /* ... yes,            */
                else
                  {
                  editName = editName.substring( 0,
                                                 editName.indexOf( ' ' ) );
                  editItem( editName,
                            fields.dumpToString( "#0", editName ),
                            fields.dumpToString( "#2", editName ) );
                  }
```

```
            }
          break;
        }
      break;
    }                              /* end switch( Where )          */
  }
/* exception occured - show error in a modal dialog window         */
catch ( Exception ex )
  {
  ex.printStackTrace();
  new AwtDialog( this,
                 "Exception",
                 AwtDialog.Show_OK,
                 new String[][][] { { { "TP", ex.toString() }  } } );
  }
}
```

When the user selects an item in the list box an Edit dialog is displayed. This dialog allows the name, type, and value to be edited. The user can also delete the item from the MQeFields object.

The same dialog is used to add a new item to the MQeFields object. In this case the item name and the value is blank, with a default type of ascii.

```
protected void editItem( String name, String type, String value )
    throws Exception
    {
    if ( fields == null )   throw new Exception( "No Fields object" );
    /* Dialog to set Field Item name type and value                 */
    AwtDialog md = new AwtDialog( this,
                                  getTitle() + " - edit item",
                                  AwtDialog.Show_OK_Cancel,
                                  new String[][][] {
        { { "L", "Field Item Name:" }, { "T", name  } },    /* Index 1 */
        { { "L", "      Data type:" }, { "D", type,         /* Index 3 */
                                               "ascii",
                                               "boolean",
                                               "byte",
                                               "double",
                                               "float",
                                               "int",
                                               "long",
                                               "short",
                                               "unicode" } },
        { { "L", "          Value:" }, { "T", value    } }, /* Index 5 */
        { { "L", "   Remove item ?" }, { "C", "Delete" } }  /* Index 7 */
                                                    } );
    /* process dialog response                                      */
    if ( md.getActionIndex( South ) == md.Button_OK )
      {
      name = md.GetText( Center, 1 );
      if ( name.equals( "" ) )
        throw new Exception( "Invalid Item name" );
      fields.delete( name );
      if ( ! md.getCheckState( center, 7 ) )   /* delete this item ?   */
        {                                     /* ... no               */
        String data = "(" + md.GetText( Center, 3 ) +
                      ")" + name +
                      "=" + md.getText( Center, 5 );
        fields.restoreFromString( "(#0)#1=#2", data );
        }
      showFields( );
      }
    }
```

This completes a functional but primitive ini file editor. It can be used to display or modify MQeMsgObjects as long as the data is not encoded.

Chapter 4. Queue managers, messages, and queues

"MQSeries Everyplace queue manager" on page 2 provides an overview of the services provided by MQSeries Everyplace queue manager, and queues. This section provides detailed descriptions of the functions and use of queue managers and their associated resources, messages and queues.

Creating and deleting queue managers

A queue manager requires at least the following:

- A registry (see "MQeRegistry parameters for the queue manager" on page 38)
- A queue manager definition
- Local default queue definitions (see "Queues" on page 60)

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.

This section provides more information to help you to use the MQeQueueManagerConfigure class.

Creating a queue manager

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

1. 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();
 // initialise the parameters
 ...
 qmConfig = new MQeQueueManagerConfigure( );
 qmConfig.activate( parms, "qmName\\Queues\\" );
}
catch (Exception e)
{ ... }
```

2. Call the constructor with parameters:

© Copyright IBM Corp. 2000

```
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialise the parameters
 ...
 qmConfig = new MQeQueueManagerConfigure( parms, "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 imbedded MQeFields object (*Name*) that contains the name of the queue manager
- An imbedded 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 (see "Aliases" on page 38), or if you wish to use an alias to set the channel attribute rule name (see "2. Set queue manager properties and create the queue manager definition" on page 33), 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();
 // initialise 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, "FileRegistry");
 regFields.putAscii(MQeRegistry.DirName, "qmname\\Registry");

 // add the imbedded MQeFields objects
 parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
 parms.putFields(MQeQueueManager.Registry, regFields);
 // set aliases
 MQe.alias("FileRegistry", "com.ibm.mqe.registry.MQeFileSession");
 MQe.alias("ChannelAttrRules", "examples.rules.AttributeRule");
 // activate the configure object
 qmConfig = new MQeQueueManagerConfigure( parms, "qmName\\Queues\\" );
}
catch (Exception e)
{ ... }
```

2. Set queue manager properties and create the queue manager definition

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 channel timeout value with **setChannelTimeout()**
- You can set the name of the channel 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();
 // initialise the parameters
 ...
 // set aliases
 MQe.alias("FileRegistry", "com.ibm.mqe.registry.MQeFileSession");
 MQe.alias("ChannelAttrRules", "examples.rules.AttributeRule");
 // activate the configure object
 qmConfig = new MQeQueueManagerConfigure( parms, "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.

3. Create definitions for the default queues

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

- An administration queue: **defineDefaultAdminQueue()**
- An administration reply queue: **defineDefaultAdminReplyQueue()**
- A dead letter queue: **defineDefaultDeadLetterQueue()**
- A default local queue: **defineDefaultSystemQueue()**

All these methods throw an exception if the queue already exists.

The administration queue and administration reply queue are needed to allow the queue manager to respond to administration messages, for example to create new connection definitions and queues.

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

The default local queue, SYSTEM.DEFAULT.LOCAL.QUEUE, has no special significance within MQSeries Everyplace itself, but it is useful when MQSeries Everyplace is used with MQSeries messaging because it exists on every MQSeries messaging queue manager.

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialise the parameters
 ...
 qmConfig = new MQeQueueManagerConfigure( parms, "qmName\\Queues\\" );
 qmConfig.setDescription("a test queue manager");
 qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
 qmconfig.defineDefaultAdminQueue();
 qmconfig.defineDefaultAdminReplyQueue();
 qmconfig.defineDefaultDeadLetterQueue();
 qmconfig.defineDefaultSystemQueue();
}
catch (Exception e)
{ ... }
```

4. 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();
 // initialise 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, "FileRegistry");
 regFields.putAscii(MQeRegistry.DirName, "qmname\\Registry");

 // add the imbedded MQeFields objects
 parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
 parms.putFields(MQeQueueManager.Registry, regFields);

 // set aliases
 MQe.alias("FileRegistry", "com.ibm.mqe.registry.MQeFileSession");
 MQe.alias("ChannelAttrRules", "examples.rules.AttributeRule");

 // activate the configure object
 qmConfig = new MQeQueueManagerConfigure( parms, "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.

Deleting a queue manager

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. You should 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. See “1. Create and activate an instance of MQeQueueManagerConfigure” on page 31.

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();
 // initialise 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();
 // initialise the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
 qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteStandardQMDefinitions();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

Starting queue managers

A queue manager can run:

- as a client
- in a server
- in a servlet

The following sections describe the example client, servers and servlet that are provided in the examples.queuemanager package. They three types of queue manager are all constructed from the same base MQSeries Everyplace components, with some additions that give each its unique properties. MQSeries Everyplace provides a 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. This is described in "Chapter 3. MQeFields" on page 19. The **loadConfigFile()** method in the MQeQueueManagerUtils class performs this function.

Client queue managers

A client typically runs on a device, and provides a queue manager that can be used by applications on the device. It can open many connections to other queue managers and, if configured with a peer channel can accept incoming requests from 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.

The following example shows a startup ini file for a typical client.

```
*
* ExamplesMQeClient.ini
*   An example ini file for a simple MQe client
*
[Alias]
*
*    Event log class
*
(ascii)EventLog=examples.log.LogToDiskFile
*
*    Network adapter class
*
(ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
*
*    Queue Manager class
*
(ascii)QueueManager=com.ibm.mqe.MQeQueueManager
*
*    Trace handler (if any)
*
(ascii)Trace=examples.trace.MQeTrace
*
*    Message Log file interface
*
(ascii)MsgLog=com.ibm.mqe.adapters.MQeDiskFieldsAdapter
*
*    Class name for File registry
*
(ascii)FileRegistry=com.ibm.mqe.registry.MQeFileSession
*
*    Class name for Private registry
*
(ascii)PrivateRegistry=com.ibm.mqe.registry.MQePrivateSession
*
*    Default Channel class
*
(ascii)DefaultChannel=com.ibm.mqe.MQeChannel
*
*    Default Transporter class
*
(ascii)DefaultTransporter=com.ibm.mqe.MQeTransporter
*
*    Channel Attribute Rules
*
(ascii)ChannelAttrRules=examples.rules.AttributeRule
*
*    Name of Base Key
*
```

```
(ascii)AttributeKey_1=com.ibm.mqe.MQeKey
*
*    Name of Shared Key
*
(ascii)AttributeKey_2=com.ibm.mqe.attributes.MQeSharedKey
*-------------------------------------------------------------------------*
*
*  Registry ( configuration data store )
*
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=FileRegistry
*
*   Location of the registry
*   (Only use relative directory for development/demo)
*
(ascii)DirName=.\ExampleQM\Registry\
*-------------------------------------------------------------------------*
*
*  Queue manager details
*
[QueueManager]
*
*   Name for this Queue Manager
*
(ascii)Name=ExampleQM
```

Aliases

The [Alias] section provides a place where aliases can be set.

The alias names are on the left of the equals sign, and the full class name is on the right. For example, the name "Trace" can now be used instead of examples.awt.AwtMQeTrace. The "(ascii)" before the alias names signifies the type of the entry, in this case as ascii string.

Aliases can be assigned for class names and other MQSeries Everyplace objects. Aliases are used by MQSeries Everyplace, and can be used by application programs, to provide a level of indirection between the application and the real object. Hence the object instance that an alias relates to can be changed without the application needing to change. This allows a configuration to be easily modified. For instance, MQSeries Everyplace provides several different trace handlers and an alias is setup for Trace that specifies which trace handler to use. Changing the Trace alias, changes the trace handler used by MQSeries Everyplace.

The alias list can include a solution's own classes.

The alias list is not processed by the queue manager itself. The queue manager requires this list to have been processed prior to its activation as several of these aliases are required to allow the queue manager to activate properly. For example, queues must have a queue store adapter defined so that they have a storage area in which to hold their messages. MsgLog is the default queue store adapter, if this is not present then a `MsgLog not found` exception is thrown.

MQeRegistry parameters for the queue manager

The Registry] section of the ini file contain type and location information for the queue manager 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
- Queue definitions
- Remote queue definitions
- Remote queue manager definitions
- User data (including configuration-dependent security information)

Registry type:

MQeRegistry.LocalRegType (ascii)
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. See "Chapter 8. Security" on page 157.

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.

File registry parameters: The following parameter is needed for a file registry:

MQeRegistry.DirName (ascii)
The name of the directory holding the registry files.

Private registry parameters: The following parameters can be used for a private registry.

MQeRegistry.DirName (ascii)
The name of the directory holding the registry files

MQeRegistry.PIN (ascii)
The PIN for the private registry

MQeRegistry.KeyRingPassword (ascii)
The password or phrase used to protect the registry's private key

MQeRegistry.CAIPAddrPort (ascii)
The address and port number of a mini-certificate issuing server

MQeRegistry.CertReqPIN (ascii)
The certificate request number preallocated by the mini-certificate administrator to allow the registry to obtain its credentials

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

For either type of registry *MQeRegistry.Separator (ascii)* is also needed if you want to use a non-default separator. The separator is the character that is used between the components of an entry name, for example:
`<QueueManager><Separator><Queue>`

This parameter is specified as a string but it should contain a *single* character. If it contains more than one only the first character is used.

You should use the same separator character every time a registry is opened. It should not be changed once a registry is in use.

If this parameter is not specified the separator defaults to "+".

Starting a client queue manager

Starting a client 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:

```
/*--------------------------------------------------------------------------*/
/* 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 manager                                                */
  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. MQSeries Everyplace and applications can use the aliases once they have been loaded. Many of the aliases shown in the ini file on page 37 are required for MQSeries Everyplace to function correctly and should not be removed.

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 MQSeries Everyplace class loader to load the class and call the null constructor.
2. Use the **activate** method to activate the queue manager 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:

```
* Extract from MQePrivateClient.ini
*
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=PrivateRegistry
*
*   Location of the registry
*
(ascii)DirName=.\ExampleQM\PrivateRegistry
*
* PIN
*
(ascii)PIN=not set
*
* Certificate request pin
*
(ascii)CertReqPIN=not set
*
* Key ring password
*
(ascii)KeyRingPassword=not set
*
* Network address of certificate authority
*
(ascii)CAIPAddrPort=9.20.7.219:8081er
```

These fields are described in “MQeRegistry parameters for the queue manager” on page 38. See “Chapter 8. Security” on page 157 for more details on secure queues and MQePrivateClient. (If you have MQSeries Standard Edition, not all security features are available.)

registry parameters

For MQePrivateClient (and MQePrivateServer) to work, the startup parameters must *not* contain *CertReqPIN*, *KeyRingPassword* and *CAIPAddrPort*. Hence the registry section for an MQePrivateClient, using the Standard Edition of MQSeries Everyplace looks like the following:

```
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=PrivateRegistry
*
*   Location of the registry
*
(ascii)DirName=.\ExampleQM\PrivateRegistry
*
* PIN
*
(ascii)PIN=not set
```

Server

A server usually runs on a gateway. 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 including servers, clients and gateways. 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 MQSeries Everyplace network . MQSeries Everyplace 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 server functions, this server can send and receive messages to and from other members of the MQSeries family. This server is in package examples.mqbridge.queuemanager and is described in "Chapter 7. MQSeries-bridge" on page 119.

Example MQeServer

MQeServer is the simplest server implementation.

This server can be started with the following command:

```
<javaCommand> examples.queuemanger.awt.MQeServer  <startupIniFile>
```

where:

javaCommand
is the command used to start Java applications (**java** for example)

startupIniFile
is an ini file that contains startup parameters for the queue manager and server (.\ExamplesMQeServer.ini for example)

The batch file ExamplesMQeServer.bat provides a shortcut for starting the server with the ini file .\ExamplesMQeServer.ini. As with the client queue manager, ini files are used to hold the server startup parameters. For a server queue manager

you must extend, the standard client queue manager ini file to include a [ChannelManager] and a [Listener section]. A typical extension to the server startup parameters follows:

```
* Extract from ExamplesMQeServer.ini
*
[ChannelManager]
*
*   Maximum number of channels allowed
*
(int)MaxChannels=0
*-------------------------------------------------------------------------*
[Listener]
*
*   FileDescriptor for listening adapter
*
(ascii)Listen=Network::8081
*
*   FileDescriptor for Network read/write
*
(ascii)Network=Network:
*
*   Channel timeout interval in seconds
*
(int)TimeInterval=300
```

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

The new sections in the ini file control channel usage. In the ChannelManager section, the *MaxChannels* parameter controls the maximum number of channels that can be open at any time. The Listener section contains parameters pertaining to how incoming network requests are handled:

Listen The network adapter that handles incoming network requests. For example this could be an http adapter or a pure tcp/ip adapter. As well as the adapter name, you can pass parameters that dictate how the adapter should listen. For instance Listen=Network::8081 means use the Network adapter where Network is an alias to listen on port 8081. (This assumes that the Network alias is set to either an http or a tcp/ip adapter.)

Network
: This parameter is used to specify the adapter to use for network read and write requests, once the initial network request has been accepted. Usually this is the same as the adapter used on the *Listen* parameter.

TimeInterval
: The time in seconds before idle channels are timed out. As channels are persistent logical entities that last longer than a single queue manager request, and can survive network breakages, it may be necessary to time out channels that have been inactive for a period of time.

The creation of MQeServer follows that of MQeClient:.

1. Pass the server startup parameters to the **init** method
2. Check to ensure that only one server will run per JVM
3. Load any aliases, and, if necessary, enable trace

The following code shows the **init** method that is used to start the server:

```
public void init( MQeFields parms ) throws Exception
{
  if ( initialised )                          /* Only one server at a time     */
    throw new Exception( "Server already running" );

  sections = parms;                           /* Remeber startup parms         */
  MQeQueueManagerUtils.processAlias( sections ); /* set any alias names       */

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

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

Once the server has been initialized, activate it using the **activate** method with a parameter of *true*. Once activated you can deactivated the server by calling the **activate** method with a parameter of *false*.

When you activate a server the following occurs:

1. The channel manager is started
2. Any additional user specified classes are loaded and the null constructor is called
3. The queue manager is started
4. The channel listener is started.

This is shown in the following code:

```
public void activate( boolean Start ) throws Exception
{
  if ( Start )                                /* activate ?                    */
  {                                           /* ... yes                       */
    if ( ! initialised )                      /* been here before              */
    {                                         /* ... no                        */
      /* allocate Chan Mgr                                                     */
      channelManager  = MQeQueueManagerUtils.processChannelManager( sections );

      /* assign any class aliases                                              */
      MQeQueueManagerUtils.processAlias( sections );

      /* check for any pre-loaded classes                                      */
      loadTable = MQeQueueManagerUtils.processPreLoad( sections );
      initialised = true;                     /* only once                     */
    }                                         /*                               */
    /* setup and activate the queue manager                                    */
    queueManager = MQeQueueManagerUtils.processQueueManager( sections,
   channelManager.getGlobalHashtable( ) );

    /* setup and activate the listener for incomming connections               */
    channelListener = MQeQueueManagerUtils.processListener(
   sections, channelManager );
  }
  else                                        /* ... no                        */
  {                                           /*                               */
    if ( channelListener != null )  channelListener.stop( );
    if ( queueManager    != null )  queueManager.close( );
    channelListener = null;                   /* release object                */
    queueManager    = null;                   /* release object                */
  }
}
```

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

When the server is deactivated:

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

The following sections of code from the MQeQueueManagerUtils class process each of the components.

The following section starts a channel manager:

```
public static MQeChannelManager processChannelManager( MQeFields sections )
throws Exception
{
  MQeChannelManager channelManager = null;    /* work variable               */
  if ( sections.contains( Section_ChannelManager ) )  /* section present ?    */
  {                                           /* ... yes                     */
    MQeFields section = sections.getFields( Section_ChannelManager );
    channelManager    = new MQeChannelManager( );     /* load the manager     */
    channelManager.numberOfChannels( section.getInt( "MaxChannels" ) );
  }                                           /*                             */
  return( channelManager );                   /* return the allocated mgr    */
}
```

This method instantiates a channel manager and then uses the *MaxChannels* parameter from the [ChannelManager] section of the startup parameters to set the maximum number of channels that are permitted.

It is also possible to specify a set of classes to load when the queue manager is loaded. These are added to a [PreLoad] section of the ini file. The entries must have the form (ascii)uniqueName=class, as shown in the following example:

```
[PreLoad]
*
*   Classes to load at server startup
*
(ascii)StartClass1=test.ServerTest1
(ascii)StartClass2=test.ServerTest2
```

The following section of code loads the preload classes:

```
public static Hashtable processPreLoad( MQeFields sections ) throws Exception
{
  Hashtable loadTable = new Hashtable();      /* allocate load table         */
  if ( sections.contains( Section_PreLoad ) ) /* section present ?           */
  {                                           /* ... yes                     */
    MQeFields section = sections.getFields( Section_PreLoad );
    Enumeration keys  = section.fields();     /* get all the keywords        */
    while ( keys.hasMoreElements() )          /* as long as there are keywords*/
    try                                       /* incase of error             */
    {                                         /*                             */
      String key = section.getAscii( (String) keys.nextElement() ).trim( );
      loadTable.put( key, MQe.loader.loadObject( key ) );
    }
    catch ( Exception e )                     /* error occured               */
    {
      e.printStackTrace();                    /* show the error              */
    }
  }
  return( loadTable );                        /* return the table            */
}
```

For each class specified in the [PreLoad] section of the ini file:

1. The class is loaded using the MQeLoader. This calls the null constructor of the class, so any initialization or/startup code must be placed in this constructor

2. Once loaded, a reference to the class is placed in a hashtable. This table can then be used by other methods in the server. For instance, the **close** method of the server could be extended to execute the **close** method of every preloaded class, when the server closes.

Example MQePrivateServer

MQePrivateServer is an extension of MQeServer with the addition that it configures the queue manager and registry to allow for secure queues. See "Chapter 8. Security" on page 157.

Example AwtMQeServer

AwtMQeServer is in package examples.awt and provides a graphical front end to the console based servers.

Start the server with the following command:

```
<javaCommand> examples.awt.AwtMQeServer <startupIniFile>
```

Where:

javaCommand
: is the command used to start Java applications (**java** for example)

startupIniFile
: is an ini file that contains startup parameters for the queue manager and server (for example .\ExamplesAwtMQeServer.ini)

Batch file ExamplesAwtMQeServer.bat provides a shortcut to start the server with the file .\ExamplesAwtMQeServer.ini.

The AwtMQeServer uses the following additional aliases:

Server the server class for which this class provides a graphical front end

Admin the name of a class that provides an administration console

The example file .\ExamplesAwtMQeServer.ini sets the aliases as follow:

```
*
*    Admin console (if any)
*
(ascii)Admin=examples.administration.console.Admin
*
*    Base Server class
*
(ascii)Server=examples.queuemanager.MQeServer
```

When the private server is started the following window is displayed:

Figure 5. AWT MQSeries Everyplace server window

The buttons function as follows:

Exit Close the server and perform a **System.exit()**

Stop | Run

If the server is running then **Stop** stops it. If the server is stopped then the button displays **Run** to start the server.

Start is performed with the following code:

```
if ( running )                    /* running ?                    */
{
  setText( North, index, "Run"  );        /* ... yes,             */
  server.activate( false );               /* stop server          */
}
else
{
  setText( North, index, "Stop" );        /* ... no,  i.e start   */
  if ( server == null )                   /* initialized before ? */
  {                                       /* yes,                 */
    /* Load the startup parms and setup class aliases             */
    MQeFields sections
      = MQeQueueManagerUtils.loadConfigFile( iniName );
    MQeQueueManagerUtils.processAlias( sections );
    /* Load the server and initialise if first pass               */
    server = (MQeServer)MQe.loader.loadObject( "Server" );
    server.init( sections );
  }
  server.activate( true );        /* Activate the server          */
}
running = ! running;              /* change state                 */
```

Trace Activates or deactivates trace. This is achieved with the following code:

```
/* Get current trace handler if any ..                           */
MQeTraceInterface    trace = MQe.getTraceHandler( );
if ( trace == null )              /* If trace is not on,start it  */
  MQeQueueManagerUtils.traceOn( this.getTitle() + " - Trace", null );
else                              /* otherwise stop it            */
  MQeQueueManagerUtils.traceOff();
```

Admin

Starts or stops the adminiistration console. The follow code implements this function:

```
if ( adminGUI != null && adminGUI.active )
{                                 /* GUI active so                */
  adminGUI.close();               /* close it                     */
  adminGUI = null;
}
else if ( adminGUI == null ||
        ( adminGUI != null && !adminGUI.active ) )
{                                 /* GUI not running or not active*/
  adminGUI = (Admin)MQe.loader.loadObject( "Admin" );
  adminGUI.activate();            /* so load and actiate it       */
}
```

Help Displays an about dialog

You can additionally turn event logging on and off and select the logger to use from the drop down list box. The following selections are possible:

- No logging,
- examples.eventlog.LogToDiskFile
- examples.eventlog.LogToNTEventLog

Servlet

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 MQSeries Everyplace components as the server, and a servlet can use a server ini file.

The main component not required in a servlet is the channel listener, this function is handled by the Web server itself. Web servers only handle http data streams so any MQSeries Everyplace client that wishes to communicate with an MQSeries Everyplace 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. The following definitions configure a connection on servlet /servlet/MQSeries Everyplace with queue manager PayrollQM:

Connection name
PayrollQM

Channel
com.ibm.mqe.MQe

Channel Adapter
com.ibm.mqe.adapters.MQe

TcpipHttpAdaper
192.168.0.10:80

Parameters
/servelet/MQe

Options

Alternatively, 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
/servelet/MQe

Options

Web servers can run multiple servlets. It is possible to run multiple different MQSeries Everyplace 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 MQSeries Everyplace servlet must run in a different Java Virtual Machine (JVM)

The MQSeries Everyplace sevlet extends javax.servlet..http.HttpServlet and overrides methods for starting, stopping and handling new requests. The following code fragment starts a servlet:

```
/**
 * Servlet Initialisation......
 */
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 channel 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 timeout from secs to millsecs)
    int tI =
      sections.getFields(MQeQueueManagerUtils.Section_Listener).getInt( "TimeInterval" );
    long timeInterval = 1000 * tI;
    channelTimer = new MQeChannelTimer( channelManager, timeInterval );

    // Servlet initialisation complete
    mqe.trace( 1300, null );
  }
  catch (Exception e)
  {
    mqe.trace( 1301, e.toString() );
    throw new ServletException( e.toString() );
  }
}
```

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 MQSeries Everyplace 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 channel listener is not started as the Web server handles all network requests on behalf of the servlet.
- As there is no channel listener a mechanism is required to timeout channels that have been inactive for longer than the timeout 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.

servlet queue managers

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 MQSeries Everyplace servlet, it passes the request to MQSeries Everyplace 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: MQSeries Everyplace only handles requests with the **doPost()** method, it does not accept requests using the **doGet()** method
2. The request is passed to MQSeries Everyplace through a channel manager. From this point, all processing of the request is handled by core MQSeries Everyplace classes such as the queue manager.
3. Once MQSeries Everyplace 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.

Configuring queue managers using base classes

Although the use of MQeQueueManagerConfigure is the recommended way to create and delete queue managers, this section describes how to create the same function from base classes.

Queue manager activation

To activate a queue manager you require:

- A pre-configured registry
- A set of activation parameters that inform the queue manager how to activate the registry

When the queue manager is activated the activation parameters are passed to it. These parameters consist of MQeFields objects imbedded inside another MQeFields object.

The names of the imbedded MQeFields objects to be used are defined in the MQeQueueManager class:

MQeQueueManager.QueueManager

The name of the queue manager being activated

MQeQueueManager.Registry

The location of the queue manager's predefined registry

MQeQueueManagerUtils.Section_Aliases
MQSeries Everyplace aliases

The registry contains the definitions of the queues that the queue manager owns, the definitions of any other queue managers known , and some configurable queue manager setup data. This setup data is:

Queue manager description
A String containing a description for the queue manager

Queue manager rules
A String containing the name of the class to use as the queue manager's Rules (see "Queue manager rules" on page 73).

Default queue store
A path name that is the location of the default queue store (where queue stores its messages). This is only used if a queue without a queue store field is added to the queue manager . The name of the queue is appended to the default string to give the queue its own unique queue store path name.

Channel attribute rules
A String containing the name of the class to use as the channel attribute rules. These rules define how to behave when dealing with remote queues that have non-null attributes .

Channel Timeout
A long value that is the channel time-out (measured in milliseconds). If a channel between two queue managers is idle for longer than this period, the channel is closed.

You can update all these values using MQSeries Everyplace administration (see "Chapter 6. Administering messaging resources" on page 83) and they can also be configured when the queue manager is created.

The MQSeries Everyplace Aliases are described in detail in "Aliases" on page 38.

MQSeries Everyplace provides two classes that start the queue manager in predefined configurations. (These classes are in the `examples` directory.)

MQeClient
Starts the queue manager as a client

MQeServer
Starts the queue manager as part of an MQSeries Everyplace server

All required processing is handled by these classes before the queue manager is started.

It is possible to process the alias list and activate the queue manager without using either of these classes. The alias list is processed using the **MQe.alias** method. In the example below, the alias name "Trace" is set to examples.awt.AwtMQeTrace.

```
alias( "Trace", "examples.awt.AwtMQeTrace" );
```

Both MQeClient and MQeServer accept an ini file containing the queue manager parameters. The entries in the ini file are converted to the required imbedded MQeFields object. This is done with the examples.queuemanager.MQeQueueManagerUtils class which makes use of the **MQe.alias** method to process the alias list.

The following code fragment shows these procedures:

```
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                   */
    {                                         /*                            */
      /* get the Keyword    */
      String key = (String) keys.nextElement();
      /* add alias          */
      MQe.alias( key, section.getAscii( key ).trim( ) );
    }                                         /*                            */
  }                                           /*                            */
}
```

The input to this method, the MQeFields object `sections`, is the ini file in MQeFields form. The ini file is converted to MQeFields object form in the **loadConfigFile()** method of MQeQueueManagerUtils (this makes use of the **MQeFields.restoreFromString()** method).

A test is made to see if `sections` contains an alias list. The alias list ini file section name is defined in the constant, `Section_Alias`. If an alias list is available, then a **getFields()** is performed on `sections`, to return the alias list (an MQeFields object). The contents of the alias list is then enumerated, and the code loops through the enumeration, calling the alias command for each alias.

Using queue managers

MQSeries Everyplace applications and the Java Virtual Machine

The Java version of the MQSeries Everyplace queue manager runs inside an instance of a Java Virtual Machine (JVM). Currently MQSeries Everyplace only allows one queue manager to be invoked per JVM. However, it is possible to invoke multiple instances of the JVM (every time the Java command is invoked a

new Java Virtual Machine is created). Hence multiple MQSeries Everyplace queue managers can be created on the same device. Each of these queue managers must have a unique name, otherwise unexpected behavior may result.

Java MQSeries Everyplace applications must run inside the same JVM as the queue manager they are using. An elegant way to do this is to use an application launcher. This is a class that starts the queue manager and a number of MQSeries Everyplace applications on separate threads. An example of such a class is shown in the following code fragment:

```
/* extends from   MQe base class */
public class appLauncher extends   MQe implements Runnable
{
  Thread[] threads    = null; /* thread references */
  String[] appList    = null; /* list of   MQSeries Everyplace apps */
  int      appCount   = 0;
  String lock = new String();
  MQeQueueManager qmgr = null; /* reference to QMgr */

  public static void main( String args[] )
  {

    try
    {
      (new appLauncher()).startApplications();
    }
    catch ( Exception e )
    {
      System.err.println( "Exception on starting applications" );
      e.printStackTrace( System.err );
    }
  }

  public void startApplications( String args[] ) throws Exception
  {
    boolean active = false; /* any active threads? */
    /* create an array of the thread references of the applications  */
    /* being launched */
    threads = new Thread[ args.length ];
    appList = args; /* keep the list of the applications to be launched */
    /* loop through the list of apps being launched & start a new  */
    /* thread for each one */
    for ( int i = 0; i < appList.length; i++ )
    {
      Thread th = new Thread( this ); /* create a new thread */
      threads[i] = th; /* keep reference */
      th.start(); /* start new thread */
      /* loop until queue manager is active then start rest of apps */
      if ( i == 0 )
        while( qmgr == null );
    }
    /* keep appLauncher thread alive until all other apps have finished */
    while( active )
    {
      active = false;
      /* loop through thread references, starting at element 1 */
      /* remember first element in appList is QMgr ini file path name */
      for( int j=1; j < appList.length; j++ )
        if ( threads[j] != null )
          active = true; /* thread still active */
    }
    if ( qmgr != null )
      qmgr.close(); /* close queue manager */
  }

  /* this method called for each application being launched, plus the  */
```

```
  /*  queue manager */
  public void run()
  {
    int currentApp; /* which element in threads table */
    synchronized( lock )
    {
      currentApp = appCount;
      appCount++; /* update count */
    }
    try
    {
      /* first element is QMgr ini file path name */
      if ( currentApp == 0 ) /* start  queue manager */
      {
        MQeClient client = new MQeClient( appList[0] );
        qmgr = client.queueManager; /* QMgr now active */
      }
      else  /* load application */
                                        /* (this invokes default constructor) */
        loader.loadObject( appList[currentApp] );
    }
    catch ( Exception e )
    {
      e.printStackTrace( System.err );
    }
    finally
    { /* get thread reference for this app */
      Thread th = threads[currentApp];
      threads[currentApp] = null; /* nullify reference */
      th.stop(); /* stop thread */
    }
  }
}
```

The arguments supplied to this class are the path name of the queue manager's ini file, followed by a list of the MQSeries Everyplace applications launch. All the applications are invoked using their default constructor.

The application launcher is started with the command

```
java appLauncher
<ini file path name><application class name><application class name>...
```

For example:

```
java appLauncher
          e:\\MQe\\TestQMgr\\TestQMgr.ini examples.queuemanager.TestMQeApp
```

All the applications should use **MQeQueueManager.getReference()** to obtain the object reference to the queue manager that is already running inside the JVM.

Launching applications with RunList

An alternative way of launching MQSeries Everyplace applications is to use the RunList mechanism. You can supply two lists of MQSeries Everyplace applications (known as *run lists*) as part of the queue manager activation parameters. The first list contains applications that are launched after the queue manager has been activated. The second list contains applications that are launched once a queue manager has received a close request.

The applications contained in the run lists should implement MQeRunListInterface. The queue manager calls the **activate()** method defined in the interface to activate the applications and pass any available setup information to it.

If an applications does not implement MQeRunListInterface, the application is just invoked and no setup information is passed to it.

The [AppRunList] section in the ini file contains the names of the applications to launch at queue manager activation time. The symbolic name of the application is on the left-hand side of the equals sign, with the full class name of the application on the right , as shown in the queue manager ini file example.

Any setup data for the application can be provided in a section headed with [symbolic name of the application].

Example queue manager ini file

```
* Sample  queue manager ini file

*  queue manager setup info
[QueueManager]
* Name for this  queue manager
(ascii)Name=ServerQMgr8082

* Registry setup info
[Registry]
* QueueManager Registry type (ascii)LocalRegType=com.ibm.mqe.registry.MQePrivateSession
* Location of the registry
(ascii)DirName=d:\development\Rename\Classes\ServerQMgr8082\Registry
* Registry access PIN
(ascii)PIN=12345678

* List of applications to launched at  queue manager activation-time
[AppRunList]
(ascii)App1=examples.queuemanager.TestMQeApp
(ascii)App2=examples.administration.AdminApp

* Setup info for App1 - the data in this section is passed to the application
[App1]
(ascii)DefaultMsgPriority = 7
(long)Timeout = 30000

* Setup info for App2 - the data in this section is passed to the application
[App1]
(ascii)DefaultQueueName=AdminReplyQueue
```

The applications that are invoked when the queue manager is activated should return control to the queue manager as quickly as possible to allow the queue manager to continue its activation. If the application is a long running task it shouldinitialize itself on a different thread from the one on which it was called. The application is responsible for the management of threads that it creates.

Applications that are invoked on queue manager **close** can block the queue manager from shutting down if they do not return.

Example of an application being launched at queue manager activation time

```
public class ExampleApp extends   MQe implements MQeRunListInterface,
                                                  Runnable,
                                                  MQeMessageListenerInterface
{
  Thread th = null;
  MQeQueueManager qmgr = null;
  ...
  /*Called by the  queue manager to activate the application          */
  public Object activate( Object owner, Hashtable loadTable,
                          MQeFields setupData )
  {
```

```
    qmgr = (MQeQueueManager)owner;   /*QMgr is owner of the application*/
    processSetupData( setupData );      /*Process the setup information*/
    th = new Thread( this );            /*Create a new thread to listen*/
    th.start();                             /*for incoming messages*/
    return (null);                     /*return control to the QMgr*/
  }

  public void run()
  {
    try
    {
       /*Create a message listener for incoming messages*/
       qmgr.addMessageListener( this, "MyQueue", null );
       /* Loop indefintely keeping application alive */
       while( true );
    }
    catch ( Exception e )
    {
      e.printStackTrace( System.err );
    }
  }
  ...
}
```

In this example, the application is invoked using the **activate()** method. This method processes its setup data, and creates a message listener on a separate thread. The application returns control to the queue manager as soon as possible, to allow the queue manager to continue its activation process. The thread that the application created remains active.

Example of an application being launched when the queue manager receives a close request

```
public class ExampleCloseApp extends   MQe implements MQeRunListInterface
{
  MQeQueueManager qmgr = null;
  ...
  /* Called by the  queue manager to activate the application  */
  public Object activate( Object owner, Hashtable loadTable,
                          MQeFields setupData )
  {
    qmgr = (MQeQueueManager)owner; /* QMgr is owner of the application */
    performAction(); /* Perform some action */
    /* don't return control to the QMgr until application has finished */
    return (null);
  }
}
```

In this example, the application is activated using its `activate()` method when the queue manager receives a `close` request. The application should not return control to the queue manager until the application has finished its processing because once the queue manager has control it continues its close-down process.

Messages

MQSeries Everyplace messages are descendant objects of MQeFields, as described in “Chapter 3. MQeFields” on page 19. Applications can data into the message as a <name, data> pairing. MQSeries Everyplace defines some constant field names that are useful to messaging applications. These are:

Unique ID
MQe.Msg_OriginQMgr + MQe.Msg_Time

Message ID
MQe.Msg_ID

Correlation ID
MQe.Msg_CorrelID

Priority
MQe.Msg_Priority

The *Unique ID* is a combination of a unique (per JVM) timestamp generated by the message object when it is created, and the name of the queue manager to which the message was first given. The *Unique ID* is used by applications to retrieve messages. It cannot be changed by an application.

The *Unique ID* uniquely identifies a message within an MQSeries Everyplace network so long as all queue managers within the MQSeries Everyplace network are named uniquely.

Note: MQSeries Everyplace does not check or enforce the uniqueness of queue manager names. It is the responsibility of an individual solution to ensure that its queue manager names are unique.

The **getMsgUIDFields()** method accesses the *Unique ID* of a message:

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

The **getMsgUIDFields()** method returns an MQeFields object that contains two fields,

- MQe.Msg_OriginQMgr
- MQe.Msg_Time

These fields can be individually retrieved as follows:

```
long timeStamp    = msgUID.getLong(MQe.Msg_Time);
String originQMgr = msgUID.getAscii(MQe.Msg_OriginQMgr);
```

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

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

The *Priority* field contains message priority values. Message priority is defined in the same way as in other members of the MQSeries family. It ranges from 9 (highest) to 0 (lowest). Applications use this field to deal with a message according to its priority.

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

Applications can create fields for their own data within messages .

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

An alternative approach is to extend MQeMsgObject to include some methods to assist with creating messages.

```
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");
  }

 /**
  * This method sets the name of the item to be ordered
  */
  public void setItem(String anItem) throws Exception
  {
    putUnicode("Item", anItem);
  }

 /**
  * This method returns the name of the item to be ordered
  */
  public String getItem() throws Exception
  {
    return getUnicode("Item");
  }

 /**
  * This method sets the quantity required
  */
  public void setQuantity(int aQuantity) throws Exception
  {
    putInt("Quantity", aQuantity);
  }

 /**
  * This method returns the quantity required
  */
  public int getQuantity() throws Exception
  {
    return getInt("Quantity");
  }

 /**
  * This method sets the name of the queue to which to send an order reply
  */
  public void setReplyToQ(String aMyReplyToQ) throws Exception
  {
    putAscii("Msg_ReplyToQ", aMyReplyToQ);
  }

 /**
```

```
  * This method returns the name of the queue to which an order reply
  * will be sent
  */
  public String getReplyToQ() throws Exception
  {
    return getAscii("Msg_ReplyToQ");
  }

 /**
  * This method sets the name of the queue manager to which an order
  * reply will be sent
  */
  public void setReplyToQMgr(String aMyReplyToQMgr) throws Exception
  {
    putAscii("Msg_ReplyToQMgr", aMyReplyToQMgr);
  }

 /**
  * This method returns the name of the queue manager to which an order
  * reply will be sent
  */
  public String getReplyToQMgr() throws Exception
  {
    return getAscii("Msg_ReplyToQMgr");
  }
}
```

The additional methods handle the **puts** and **gets** of the data in and out of the message object. Application programmers do not need to be involved with either the type of the data being sent, or the field names being used inside the message.

```
OrderRequestMsg orderRequest = new OrderRequestMsg();
orderRequest.setClientNo( 1234 ); /* client ref. number */
orderRequest.setItem( "  MQSeries Everyplace Programmers Guide" ); /* item being ordered */
orderRequest.setQuantity( 12 ); /* quantity */
/* send the order reply to QMgr1.OrderReplyQueue */
orderRequest.setReplyToQMgr( "QMgr1" );
orderRequest.setReplyToQ( "OrderReplyQueue" );
```

Filters

The concept of filters allows MQSeries Everyplace to perform powerful message searches. Most of the major queue manager operations support the use of filters. You can create filters by placing fields into MQeFields objects. For example, if a simple **get** message operation takes a "null" filter, the result of the operation is the return of the first available message on the queue.

```
qmgr.getMessage( "myQMgr", "myQueue", null, null, 0 );
```

The use of a filter causes an application to return the first available message that contains the same fields and values as the filter. For example, the following code creates a filter that obtains the first message with a message id of "1234":

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

The filter is passed into the **get** message operation:

```
qmgr.getMessage( "myQMgr", "myQueue", filter, null, 0 );
```

Message index fields

MQSeries Everyplace stores its messages in the persistent store provided by each queue, (unless the application is using a memory queue store adapter). It does not hold the entire message in memory due to memory size constraints. Instead, to

enable faster message searching, MQSeries Everyplace holds specific fields from each message in a *message index*. The fields that are held in the index are:

Unique ID
MQe.Msg_OriginQMgr + MQe.Msg_Time

Message ID
MQe.Msg_ID

Correlation ID
MQe.Msg_CorrelID

Priority
MQe.Msg_Priority

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

Message Expiry

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 marked as expired. The queue rules then decide what happens to the message once it has been marked as expired.

Messages can also have an expiry interval themselves. You can define this by adding an 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).

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 );
/* put message onto queue */
qmgr.putMessage( null, "MyQueue", msgObj, null, 0 );
```

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 );
```

Queues

Queue managers manage queues that hold messages. The queue entities are not directly visible to an application and all interactions with the queues take place through queue managers. Each queue manager has the ability to have queues that it manages and owns. These queues are known as *local* queues. MQSeries

Everyplace also allows applications to access messages on queues that belong to another queue manager. These queues are known as *remote* queues. The same sets of operations are available on both local and remote queues, with the exception of defining message listeners (see "Message listeners" on page 70).

The messages on the queues are held in the queue's persistent store. A queue accesses its persistent store through a queue store adapter. (See "Chapter 10. MQSeries Everyplace adapters" on page 205). These adapters are interfaces between MQSeries Everyplace 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. The backing store used by a queue can be changed using an MQSeries Everyplace 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 MQSeries Everyplace to assure the delivery of messages.

Queue types

The MQSeries Everyplace queue types are described briefly in "MQSeries Everyplace queues" on page 3, and more detailed information is provided in "Local queue" on page 99.

Queue ordering

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 all the messages on a queue

When a queue is empty, the queue throws an `Except_Q_NoMatchingMsg` exception if a **get** message command is issued. This allows you to create an application that reads all the available messages on a queue.

By encasing the **getMessage()** call inside a `try..catch` block, you can 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 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;
}
```

Note: This function is not supported on MQSeries-bridge queues.

Synchronous and asynchronous messaging

MQSeries Everyplace allows flexibility in the way that applications process their messages. Messages can be transmitted *synchronously* or *asynchronously*.

Synchronous messaging

An application does not need to know how or when its messages are transmitted, however it can take control of this process if it wishes, using synchronous messaging. Synchronous messaging means that the message is transmitted as soon as the **put** message command is issued. This type of messaging can only take place when both local and target queue managers are online simultaneously, and does not work if the queue manager is not connected to the network. Synchronous messaging offers the performance advantages of instant connection and the knowledge that a message has reached its destination.

Asynchronous messaging

Asynchronous messaging allows an application to continue processing messages, whether or not the device is connected to a network. The application puts a message to a remote queue, and the message is stored by the queue manager. The message is transmitted later when a connection is established to the remote queue manager. The application does not need to be aware of when the transmission takes place.

The typical example of asynchronous messaging is an application for a field engineer or salesman. The field personnel can send orders or inventories when it is convenient. The messages are stored locally until the device is physically connected to a network. When a connection is made, the messages can be transmitted.

For asynchronous transmission to occur, the queue manager must be *triggered*. The triggering is done either by an application calling the queue manager's **triggerTransmission()** method, or by using the queue manager's transmission rules (see "Transmission Rules" on page 74). The method of message transmission depends on how the remote queue is defined. A queue manager that is sending a message to a remote queue holds a definition of that queue. This definition is known as a *remote queue definition*. When a message is put to a remote queue, the local queue manager determines how to transmit the message using the remote queue definition.
Messages are transmitted from the local queue manager to the remote queue

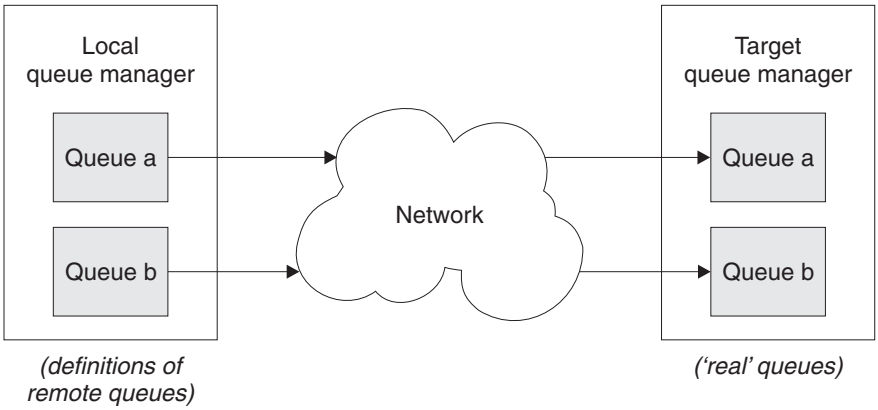

Figure 6. MQSeries Everyplace message flow

manager using the authenticator, cryptor, and compressor that are defined on the remote queue. Before it can create a message channel between the two queue

managers, the local queue manager needs to know the remote queue attributes. The local queue manager keeps this information as part of its remote queue definition.

The two transmission styles handle this differently.

If an application puts a message to a remote queue and a definition of the remote queue is held locally then the remote queue definition is used to determine characteristics of the queue. If a definition is not held locally, *queue discovery* occurs. This local queue manager synchronously contacts the remote queue manager in an attempt to ascertain characteristics of the queue. The following characteristics are discovered:

- Queue_Description
- Queue_Expiry
- Queue_MaxQSize
- Queue_MaxMsgSize
- Queue_Priority
- Queue_Cryptor
- Queue_Authenticator
- Queue_Compressor
- Queue_TargetRegistry
- Queue_AttrRule

After successful discovery of a queue, the definition of the queue is stored as a remote queue definition on the queue manager that initiated the discovery. This discovered queue definition is treated like a normal remote queue definition. The *Queue_Mode* is not discovered as all discovered queues are set for synchronous operation.

Asynchronous transmission is not able to request information from the target queue manager. Therefore, a remote queue definition must exist before asynchronous transmission can occur. Remote queue definitions can be created using MQSeries Everyplace administration messages (see "Chapter 6. Administering messaging resources" on page 83).

The combination of synchronous and asynchronous messaging allows MQSeries Everyplace to cope with unreliable communications links. If a message cannot be delivered immediately because a link is down, then the message can be queued for delivery at a subsequent time. An example of this is shown below. By defining two queues the application can handle a situation where synchronous transmission is not possible.

```
try
{
  qmgr.putMessage( "RemoteQMgr", "TransactionQueue", msgObj, null, 0 );
}
catch ( Exception e )
/* reset message UID */
msgObj.resetMsgUIDFields();
{ /* if connection cannot be made, put message on asynchronous queue */
  if ( e.getMessage().equals( "Connection Refused" )
    qmgr.putMessage( "RemoteQMgr", "AsynchTransactionQueue",
                     msgObj, null, 0 );
}
```

Browse and Lock

Browsing a group of messages and locking them allows an application to assure that no other application is able to process the messages while they are locked. The messages remain locked until they are unlocked by the application. No other application can unlock the messages.

```
MQeEnumeration msgEnum = qmgr.browseMessagesAndLock( null, "MyQueue", null,
                                                     null, 0, false );
```

This command locks all the messages on the queue `MyQueue` that exists on the local queue manager (null is an alias for the local queue manager). These messages can now only be accessed by the application that locked them. (Any messages arriving on the queue after the **Browse and Lock** operation will not be locked).

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:

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

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 *lock ID* that is returned with the enumeration of messages. Specifying the *lock ID* allows applications to work with locked messages without having to unlock them first. The following code performs a **delete** on all the messages returned in the enumeration. The message's *Unique ID* and *lock ID* 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 );
}
```

As an alternative to using the standard **java.util.Enumeration nextElement()** method, MQeEnumeration supplies a **getNextMessage()** method. This method works differently depending upon the *justUID* parameter of the **browseMessages()** method. This parameter determines whether the browse operation returns all the fields contained within the messages that it matches, or just the *Unique ID* field.

If the *justUID* parameter is set to `false`, the MQeEnumeration returned by the browse contains all the fields from the matching messages. In this case, the **getNextMessage()** method works like **nextElement()**.

If the *justUID* parameter is set to `true`, the MQeEnumeration returned by the browse contains only the *Unique ID* (`MQe.Msg_OriginQMgr` and `MQe.Msg_TimeStamp`) fields of the matching messages. In this case a proper **get** message is performed and the message is removed from the queue.

Assured message delivery can be used when getting the message. Specifying a nonzero *confirm ID* means that a confirmation of the get is required (for details of assured message delivery, see "Assured message delivery").

Instead of removing the messages from the queue, it is also possible just to unlock them, this makes them visible once again to all MQSeries Everyplace applications. You can achieve this by using the **unlockMessage()** method.

Note: This function is not supported on MQSeries-bridge queues.

Assured message delivery

Asynchronous transmission introduces the concept of *assured message delivery*. When delivering messages asynchronously, MQSeries Everyplace guarantees to deliver that 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 be lost, but remains stored on the local queue manager.

Synchronous assured message delivery

Put message

You can perform assured message delivery using synchronous message transmission, but the application must take responsibility for error handling.

Non-assured delivery of a message takes place in a single network flow. The queue manager sending the message creates a channel to the destination queue manager, and attaches a transporter to that channel. The transporter points to the destination queue. (A suitable channel and transporter may exist from a previous operation, if so they are used instead).

The message to be sent is dumped to create a byte-stream, and this bye 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 MQSeries Everyplace 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 MQSeries Everyplace 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.

The *confirmId* parameter of the **putMessage** method dictates whether the confirm flow is sent or not. A value of `zero` means that message transmission occurs in one flow, while a `nonzero` value 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 MQSeries Everyplace applications, until the confirm flow is received.

An MQSeries Everyplace 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 MQSeries Everyplace applications. You can confirm only one message at a time, it is not possible to confirm a batch of messages.

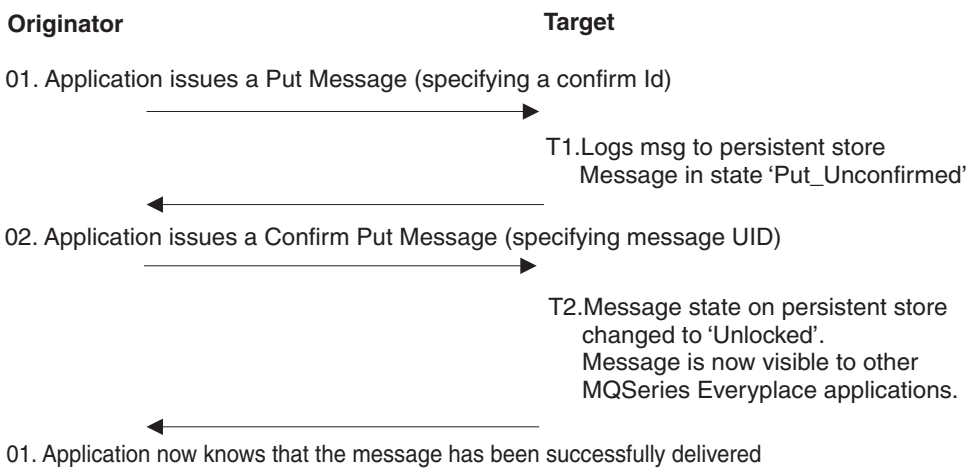

Figure 7. Assured put of synchronous messages

The **confirmPutMessage()** method requires you to specify the *UID* of the message, not the *Confirm ID* used in the prior put message command. (The *Confirm ID* is used to restore messages that remain locked after a transmission failure. This is explained in detail on page 69.)

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 */
}
```

If a failure occurs during step 1 in Figure 7 the application should retransmit the message. There is no danger of introducing duplicate messages into the MQSeries Everyplace 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 MQSeries Everyplace 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 Figure 7 on page 66 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.

The code below is taken from examples.application.example6

```
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 ++;
  }
}
```

Get message

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.

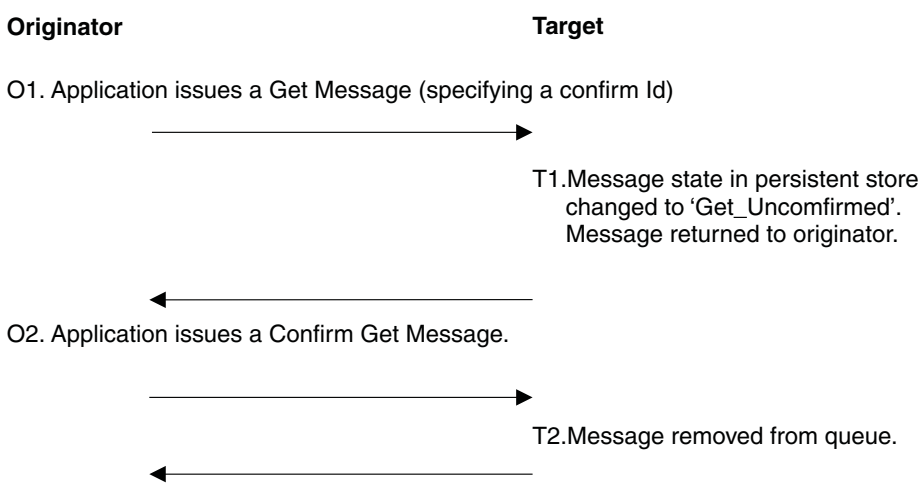

Figure 8. Assured get of synchronous messages

The following code is taken from `examples.application.example6`

```
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 */
  }
}
```

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 the **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 MQSeries Everyplace 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.

```
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 */
  }
}
```

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.

Message listeners

MQSeries Everyplace allows an application to *listen* for events occurring on queues. The notification takes the form of a standard Java event, and the listening application implements an interface that provides methods that are called when an event occurs. The application is able to specify message filters to identify the messages in which it is interested.

```
/* 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( "MyQueue", this, filter );
```

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

- The name of the queue on which to listen for message events
- 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.

An example of the way in which an application would normally handle message events is given below.

```
public void messageArrived( MQeMsgEvent msgEvent )
{
  String queueName = msgEvent.getQueueName()
  if ( queueName.equals( "MyQueue"))
  {
    /* get message from queue */
    MQeMsgObject msg = qmgr.getMessage( null, queueName,
                                        msgEvent.getMsgUIDFields(), null, 0 );
    /* ...and process it */
    processMessage( msg );
  }
}
```

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 *message ID*
- The *correlation ID*
- 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

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.

Messaging operations

Table 2 shows the operations that can be performed on messages on the various queue types.

Table 2. Messaging operations

Operation	Local queues	Remote queues	
		Synchronous	**Asynchronous**
Browse(&Lock)	yes	yes	
delete	yes	yes	
get	yes	yes	
listen	yes		
put	yes	yes	yes
wait	yes	yes	

Security

The queue manager fully supports the security functions supplied with MQSeries Everyplace. Any messages stored in a queue defined with security characteristics are encoded using those characteristics. Any communication channels set up between a queue manager and a secure queue use the security characteristics of the queue, or an existing channel with equal or higher security.

security

Messages can be individually protected by attaching security characteristics to them directly. The correct characteristics must be presented whenever dealing with a message protected in this manner.

See “Chapter 8. Security” on page 157 for a detailed discussion of MQSeries Everyplace security.

Chapter 5. Rules

MQSeries Everyplace uses the concept of *rules* to govern the behavior of its major components. Rules allow a solution to have some control over the internal workings of MQSeries Everyplace. Rules take the form of Java classes that are loaded by MQSeries Everyplace components when they are initialized.

A component's rules are called at certain points during the component's execution cycle. The component expects a method with a particular signature to be available, so when producing an extension of the base rules, care must be taken to use the correct method signatures.

Default or example rules are provided for all MQSeries Everyplace components, however it is expected that a solution would provide its own rules to customize MQSeries Everyplace behavior to fit the solution requirements.

Queue manager rules

Queue manager Rules are called when:

- The queue manager is activated
- The queue manager is closed
- A queue is added to the queue manager
- A queue is removed from the queue manager
- 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 (Transmission Rules)
- An incoming peer connection is established

Using queue manager rules

This section describes some examples of the use of the queue manager rules.

The first example shows a **put message** rule that insists that any message being put to a queue using this queue manager must contain an MQSeries *message ID* field.

```
/* 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" );
}
```

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.

© Copyright IBM Corp. 2000

```
/* 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" );
    }
  }
}
```

This previous rule is a simple example of protecting a queue. However, for more comprehensive security, you are recommended to use of an *authenticator*. An authenticator allows an application to create access control lists, and to manage who is able to get messages from queues.

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 queue in question. In the following 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.

```
/* This rule prevents the removal of the Payroll Queue */
public void removeQueue( MQeQueue queue ) throws Exception
{
  if ( queue.getQueueName().equals( "PayrollQueue" ) )
    throw new MQeException( Except_Rule, "Can't delete this queue" );
}
```

A queue manager can define its own peer channel listener. The listener detects incoming connection attempt from other queue managers made through a peer channel. The following rule is called whenever a connection request is detected. The rule is passed the name of the queue manager that is trying to connect.

```
public void peerConnection( String qmgrName )
{
  /* block any connection attempt from 'RogueQMgr' */
  if ( qmgrName.equals( "RogueQMgr" ) )
    throw new MQeException( Except_Rule, "Connection not allowed" );
}
```

Transmission Rules

A message that is put to a remote queue and 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, but the process can also be controlled by the queue manager's transmission rules.

The transmission rules are a subset of the queue manager rules.

There are two methods within the rules class that allow control over message transmission:

triggerTransmission()
Determines whether to allow message transmission at the time when the rule is called

transmit()
Makes a decision to allow transmission for each individual queue. 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

MQSeries Everyplace calls the **triggerTransmission** rule when a message is put onto a remote asynchronous queue. The queue manager **triggerTransmission** method overrides this rule and causes an attempt to transmit any pending messages

```
/* default 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". So, the above rule attempts to transmit all messages immediately. This is the default **triggerTransmission()** rule contained in the base queue manager rules class com.ibm.mqe.MQeQueueManagerRule. The rule attempts to transmit a message as soon as it is put onto a queue. This near-synchronous mode of operation is inefficient, since it sends all messages individually. It is usually advantageous to send groups of messages to utilize the network more efficiently.

A more complex rule could decide whether or not to transmit immediately based on the priority of the message. The following example shows a rule that triggers the queue manager if a message arrives that has a priority greater than 5.

```
/* Decide to transmit based on priority of message */
public boolean triggerTransmission( int noOfMsgs, MQeFields msgFields )
{
  if ( msgFields == null ) /* msg fields may be null */
    return false;
  if ( !(msgFields.contains( MQe.Msg_Priority )) )
    return false; /* no priority field in message */
  byte priority = msg.GetByte( MQe.Msg_Priority );
  if ( priority > 5 ) /* if message priority greater than 5 */
    return true; /* then transmit */
  else
    return false; /* else do not transmit */
}
```

The *msgFields* parameter contains selected fields from the message. These fields are:

- *Unique ID*
- *Message ID*
- *Correlation ID*
- *Priority*

If the rule decides to allow transmission, then all pending messages are transmitted, not just the message that was put to the asynchronous remote queue.

The *noOfMsgs* parameter contains the number of messages that are awaiting transmission. A solution may decide to implement a rule that blocks transmission until a certain number of messages are pending. Such a rule helps to make more efficient use of the network connection.

The rule below blocks until at least 10 messages are awaiting transmission.

```
public void triggerTransmission( int noOfMsgs, MQeFields msgFields )
{
  if ( noOfMsgs >= 10 ) /* if more than 10 msgs are waiting */
    return true; /* then transmit */
  else
    return false;
}
```

Transmit rule

The **transmit()** rule is only called if the **triggerTransmission()** rule allows transmission, (returns a value of `true`). The **transmit()** rule is called for every remote queue definition that holds messages awaiting transmission. This means that the rule can decide which messages to transmit from each queue.

The 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( MQeQueue queue )
{
  if ( queue.getDefaultPriority() > 5 )
    return (true);
  else
    return (false);
}
```

A sensible extension to this rule would be to allow all messages to be transmitted at 'off-peak' time. This would cause only messages from high-priority queues to be transmitted during peak periods. The following examples, show rules that implement similar ideas.

The following example only allows messages to be transmitted if the queue contains more than 10 messages.

```
public boolean transmit( MQeQueue queue )
{
  if ( queue.getNumberOfMessages() >= 10 )
    return (true);
  else
    return (false);
}
```

The following more complex 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 10 minutes                        */
  public final int triggerInterval   = 600000;
  /* cheap rate transmission period start and end times        */
  public final int cheapRatePeriodStart = 18; /* 18:00 hrs        */
  public final int cheapRatePeriodEnd   = 9; /* 09:00 hrs        */

  /* background thread reference        */
  protected Thread    th                 = null;
}
```

The constants `cheapRatePeriodStart` and `cheapRatePeriodEnd` define the extent of this cheap rate period. In this example, the cheap-rate peeriod is defined as being between 18:00 hours in the evening until 09:00 hours the following morning.

```
/* cheap rate transmission period start and end times         */
public final int cheapRatePeriodStart = 18; /* 18:00 hrs */
public final int cheapRatePeriodEnd = 9; /* 09:00 hrs */
```

The constant `triggerInterval` defines the period of time (in milliseconds) between each triggering of the queue manager.

```
public final int triggerInterval = 600000;
```

In this example, the trigger interval is defined to be 600,000 milliseconds, which is equivalent to 600 seconds, or 10 minutes.

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()** rule 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()
  {
    /* background thread which triggers XmitQ */
    th = new Thread( this );
    th.start();                     /* start timer thread */
  }
 /**
  * Overrides MQeQueueManagerRule.queueManagerClose()
  * Stops the timer thread
  */
  public void queueManagerClose()
  {
    th.stop();                      /* stop timer thread                   */
  }
```

The code to handle the background thread looks like this:

```
/**
 * Timer thread
 * Triggers queue manager every interval until thread is stopped
 */
 public void run()
 {
   try
   {
     while ( true )
     { /* sleep for specified interval                                    */
       Thread.sleep( triggerInterval );
       /* if cheap rate period call queue manager to trigger transmission */
       if ( timeToTransmit() )
         ((MQeQueueManager)owner).triggerTransmission();
     }
   }
   catch ( Exception e )
   {
     e.printStackTrace( System.err );
   }
 }
```

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 (remember it is stopped by the **queueManagerClose()** rule), and it sleeps until the end of the trigger interval period. At the end of the trigger interval, 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 */
  long currentTimeLong = System.currentTimeMillis();

  Date date = new Date( currentTimeLong );
  Calendar calendar = Calendar.getInstance();
  calendar.setTime( 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 */
}
```

Activating asynchronous remote queue definitions

The queue manager can activate its asynchronous remote queue definitions at startup time. Activating the queues means that an attempt is made to transmit any messages they contain. This behavior is configurable with the **activateQueues()** rule.

The basic rule just returns `true` or `false`.

```
public boolean activateQueues()
{
  return true; /* always transmit on activate */
}
```

Like other rules, a check can be made to see if the current time is inside the cheap-rate transmission period.

```
public boolean activateQueues()
{
  if ( timeToTransmit() )
    return true;
  else
    return false;
}
```

This rule also determines whether home-server and store-and-forward queues are activated at startup time.

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()** rule.

Queue rules

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.

Queue rules are called when:

- The queue is activated
- The queue is closed
- A message is placed on the queue (put)
- A message is removed from the queue (get)
- A message is deleted from the queue (delete)
- The queue is browsed
- An undo operation is performed on a message on the queue
- A message listener is added to the queue
- A message listener is removed from the queue
- A message expires
- When the queue's use count changes
- When an attempt is made to change a queue's attributes (authenticator, cryptor, compressor)
- An index entry is created for a message

Index entry rule

The queue does not hold all its messages in memory. They are saved into the queue store, and restored to memory when required. The queue maintains an index entry for each message held in its queue store. The index entry contains state information for the message, such as whether it is locked or unlocked. Also, certain fields from the message, known as *index fields* are stored in the index entry. The default index fields are message *`Unique ID`*, *`Message ID`*, *`Correlation ID`*, and message *`priority`*. These fields are stored because they are present in most messages, and storing them in memory allows for faster message searching.

The **indexEntry()** rule is called whenever an index entry is created. This occurs whenever a new message is put onto the queue, or at queue activation time, when the queue reads any messages left in its queue store from a previous session. The rule allows a solution to alter the index entry when it is created. A use for this would be to add commonly-used fields into the index, to improve message search times.

```
/* if the message contains a customer number field - then add this field */
/* to the message's index entry.  */
/* This will enable faster message searching */
public void indexEntry( MQeFields entry,
                        MQeMsgObject msg ) throws Exception
{
  if ( msg.contains( "Cust_No" ) )
    entry.copy( msg, true, "Cust_No" );
}
```

The parameter, *entry* contains a blank index entry for the message.The default index fields are added by the queue, after the **indexEntry** rule returns. In the previous example, if the message contains a field named *Cust_No* this is added to the message's index entry.

In subsequent messaging operations, such as get or browse, the application can use the *Cust_No* field as part of the filter supplied to the operation. Imagine that the application wants to find a message containing a *Cust_No* field with a value of "75", and an *Order_No* field with a value of "115". The queue can check the index entries and load only messages containing a *Cust_No* field with a value of "75" into memory, to see if they contain an *Order_No* field with the correct value. If the *Cust_no* field is not part of the index, every message is loaded into memory to check if it contains fields that match the filter.

Of course, the use of index fields is a compromis., They can be used to speed message search times, but they are held in memory, which may be at a premium on a pervasive device.

Message Expired rule

Both queues and message 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. This rule determines what happens to the message. Typically the message is either deleted, or placed on a dead-letter queue. However, the rule can decide to do something different. For example it can leave the message intact on the queue so that it remains visible to MQSeries Everyplace applications.

```
/* 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(
                                    ((MQeQueue)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 & the message will be deleted from the queue */
  return (true);
}
```

The previous 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`. It is worth noting that 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 MQSeries Everyplace network. So, before moving the message to the dead-letter queue, he rule must set the resend flag. This is done by adding the *MQe.Msg_Resend* field to the message. The message expiry time field must be deleted before moving the message to the dead-letter queue.

Returning a value of `true` informs the queue that the rule has determined that the message has expired.

Logging an add message listener event

The following example shows how to log an event that occurs on the queue. In the example the event that occurs is the creation of a message listener, but the principal can be used for any other queue event such as a put message, or browse message request.

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_Queue_Activate, "Queue " +
         ((MQeQueue)owner).getQueueManagerName() + " + " +
         ((MQeQueue)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_Queue_Closed, "Queue " +
         ((MQeQueue)owner).getQueueManagerName() + " + " +
         ((MQeQueue)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 " +
       ((MQeQueue)owner).getQueueManagerName() + "+" +
       ((MQeQueue)owner).getQueueName() );
}
```

Chapter 6. Administering messaging resources

The administration of MQSeries Everyplace resources such as queue managers and queues is performed using specialized MQSeries Everyplace messages. Using messages allows administration to be performed locally or remotely.

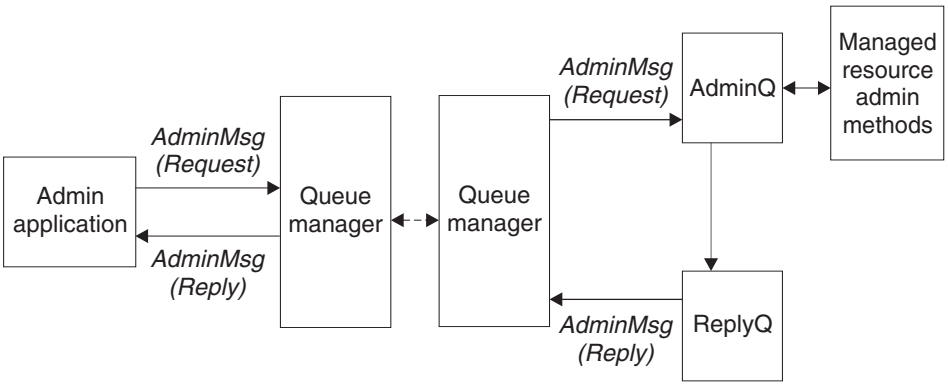

Figure 9. MQSeries Everyplace administration

Before you can administer a queue manager or its resources, you must start the queue manager configure an administration queue configured on it. The administration queue's role is to process administration messages in the sequence that they arrive on the queue. Only one request is processed at a time. The queue can be created using the **defineDefaultAdminQueue()** method of the MQeQueueManagerConfigure class. The name of the queue is `AdminQ` and applications can refer to it using the constant `MQe.Admin_Queue_Name`.

A typical administration application instantiates a subclass of MQeAdminMsg, configures it with the required administration request, and passes it to the `AdminQ` on the target queue manager. If the application wishes to know the outcome of the action, a reply can be requested. When the request has been processed the result of the request is returned in a message to the reply-to queue and queue manager specified in the request message.

The reply can be sent to any queue manager or queue but you can configure a default reply-to that is used solely for administration reply messages. This default queue is created using the **defineDefaultAdminReplyQueue()** method of the MQeQueueManagerConfigure class. The name of the queue is `AdminReplyQ` and applications can refer to it using the constant `MQe.Admin_Reply_Queue_Name`

The administration queue does not understand how to perform administration of individual resources. This knowledge is encapsulated in each resource and its corresponding administration message. The following messages are provided for administration of MQSeries Everyplace resources:

© Copyright IBM Corp. 2000

administration

Table 3. Administration messages

Message name	purpose
MQeAdminMsg	an abstract class that acts as the base class for all administration messages
MQeAdminQueueAdminMsg	provides support for administering the administration queue
MQeConnectionAdminMsg	provides support for administering connections between queue managers
MQeHomeServerQueueAdminMsg	provides support for administering home-server queues
MQeQueueAdminMsg	provides support for administering local queues
MQeQueueMangerAdminMsg	provides support for administering queue managers
MQeRemoteQueueAdminMsg	provides support for administering remote queues
MQeStoreAndForwardQueueAdminMsg	provides support for administering store-and-forward queues
MQeMQBridgeQueueAdminMsg	provides support for administering a queue that connects to an MQSeries system

These base administration messages are provided in the com.ibm.mqe.administration package. Other types or resource can be managed by subclassing either MQeAdminMsg or one of the existing administration messages. For instance, an additional administration messages for managing the MQSeries-bridge, are provided in the com.ibm.mqe.mqbridge package.

The basic administration request message

Every request to administer an MQSeries Everyplace resource takes the same basic form. Figure 10 on page 85shows the basic structure for all administration request messages:

A request is made up of:

1. Base administration fields, that are common to all administration requests
2. Administration fields, that are specific to the resource being managed
3. Optional fields to assist with the processing of administration messages

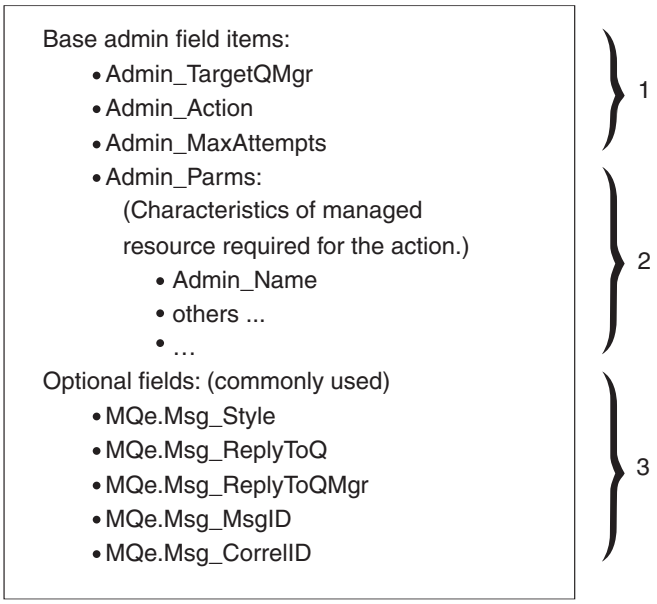

Figure 10. Administration request message

Base administration fields

The base administration fields, that are common to all administration messages, are:

Admin_Target_QMgr
: This field provides the name of the queue manager on which the requested action is to take place (target queue manager). The target queue manager can be either a local or a remote queue manager. As only one queue manager can be active at a time in a Java Virtual Machine, the target queue manager, and the one to which the message is put, are the same.

Admin_Action
: This field contains the administration action that is to be performed. Each managed resource provides a set of administrative actions that it can perform. A single administration message can only request that one action be performed. the following common actions are defined:

Table 4. Administration actions

Administration action	**Purpose**
Action_Create	Create a new instance of a managed resource.
Action_Delete	Delete an existing managed resource
Action_Inquire	Inquire on one or more characteristics of a managed resource
Action_InquireAll	Inquire on all characteristics of a managed resource
Action_Update	Update one or more characteristics of a managed resource

All resources do not necessarily implement these actions. For instance, it is not possible to create a queue manager using an administration message. Specific administration messages can extend the base set to provide additional actions that are specific to a resource.

Each common action provides a method provided that sets the Admin_Action field:

Table 5. Setting the administration action field

Administration action	Setting method
Action_Create	create (MQeFields parms)
Action_Delete	delete(MQeFields parms)
Action_Inquire	inquire(MQeFields parms)
Action_InquireAll	inquireAll(MQeFields parms)
Action_Update	update(MQeFields parms)

Admin_MaxAttempts

This field determines how may times an action can be retried if the initial action fails. The retry occurs either the next time that the queue manager restarts or at the next interval set on the administration queue.

Other fields

For most failures further information is available in the reply message. It is the responsibility of the requesting application to read and handle failure information. See "The basic administration reply message" on page 89 for more details on using the reply data.

A set of methods are available for setting some of the request fields:

Table 6. Setting administration request fields

Administration action	field type	set and get methods
Admin_Parms	MQeFields	MQeFields getInputFields()
Admin_Action	int	setAction (int action)
Admin_TargetQMgr	ascii	setTargetQMgr(String qmgr)
Admin_MaxAttempts	int	setMaxAttempts(int attempts)

Fields specific to the managed resource

Admin_Parms

This field contains the resource characteristics that are required for the action.

Every resource has a set of unique characteristics. Each characteristic has a name, type and value, and the name of each is defined by a constant in the administration message. The name of the resource is a characteristic that is common to all managed resources. The name of the resource is held in the *Admin_Name*, and it has a type of ascii.

The full set of characteristics of a resource can be determined by using the **characteristics()** method against an instance of an administration message. This method returns an MQeFields object that contains one field for each characteristic. MQeFields methods can be used for enumerating over the set of characteristics to obtain the name, type and default value of each characteristic.

The action requested determines the set of characteristics that can be passed to the action. In all cases, at least the name of the resource, *Admin_Name*, must be passed. In the case of **Action_InquireAll** this is the only parameter that is required.

The following code could be used to set the name of the resource to be managed in an administration message:

```
SetResourceName( MQeAdminMsg msg, String name )
{
  MQeFields parms;
  if ( msg.contains( Admin_Parms ) )
    parms = msg.getFields( Admin_Parms );
  else
    parms = new MQeFields();

  parms.putAscii( Admin_Name, name );
  msg.putFields( Admin_Parms, name );
}
```

Alternatively, the code can be simplified by using the **getInputFields()** method to return the *Admin_Parms* field from the message, or **setName()** to set the *Admin_Name* field into the message. This is shown in the following code:

```
SetResourceName( MQeAdminMsg msg, String name )
{
  msg.SetName( name );
}
```

Other useful fields

By default, no reply is generated, when an administration request is processed. If a reply is required, then the request message must be setup to ask for a reply message. The following fields are defined in the MQe class and are used to request a reply.

Msg_Style
: A field of type int that can take one of three values:

  Msg_Style_Datagram
  : A command not requiring a reply

  Msg_Style_Request
  : A request that would like a reply

  Msg_Style_Reply
  : A reply to a request

If *Msg_Style* is set to Msg_Style_Request (a reply is required) then the location that the reply is to be sent to must be set into the request message. The two fields used to set the location are:

Msg_ReplyToQ
: An ascii field used to hold the name of the queue for the reply

Msg_ReplyToQMgr
: An ascii field used to hold the name of the queue manager for the reply

If the reply-to queue manager is not the queue manager that processes the request then the queue manager that processes the request must have a connection defined to the reply-to queue manager.

For an administration request message to be correlated to its reply message the request message needs to contain fields that uniquely identify the request, and that can then be copied into the reply message. MQSeries Everyplace provides two fields that can be used for this purpose:

Msg_MsgID
A byte array containing the message ID

Msg_CorrelID
A byte array containing the Correl ID of the message

Any other fields can be used but these two have the added benefit that they are used by the queue manager to optimize searching of queues and message retrieval. The following code fragment provides an example of how to prime a request message:

```
public class LocalQueueAdmin extends MQe
{
  public String   targetQMgr = "ExampleQM";   // target queue manager

public MQeFields primeAdminMsg(MQeAdminMsg msg) throws Exception
{
  /*
   * Set the target queue manager that will process this message
   */
  msg.setTargetQMgr( targetQMgr );

  /*
   * Ask for a reply message to be sent to the queue
   * manager that processes the admin request
   */
  msg.putInt  (MQe.Msg_Style,       MQe.Msg_Style_Request);
  msg.putAscii(MQe.Msg_ReplyToQ,    MQe.Admin_Reply_Queue_Name);
  msg.putAscii(MQe.Msg_ReplyToQMgr, targetQMgr);

  /*
   * Setup the correl id so we can match the reply to the request.
   * - Use a value that is unique to the this queue manager.
   */
  byte[] correlID = Long.toHexString( (MQe.uniqueValue()).getBytes() );
  msg.putArrayOfByte( MQe.Msg_CorrelID, correlID );

  /*
   * Ensure matching response message is retrieved
   * - set up a fields object that can be used as a match parameter
   *   when searching and retrieving messages.
   */
  MQeFields msgTest = new MQeFields();
  msgTest.putArrayOfByte( MQe.Msg_CorrelID, correlID );

  /*
   * Return the unique filter for this message
   */
  return msgTest;
}
```

When the administration request message has been created, it is sent to the target queue manger using standard MQSeries Everyplace message processing APIs. Depending on how the destination administration queue is defined, delivery of the message can be either synchronous or asynchronous.

Standard MQSeries Everyplace message processing APIs are also used to wait for a reply, or notification of, a reply. There is a time lag between sending the request and receiving the reply message. The time lag may be small if the request is being processed locally or may be long if both the request and reply messages are delivered asynchronously. The following code fragment could be used to send a request message and wait for a reply:

```
public class LocalQueueAdmin extends MQe
{
  public String   targetQMgr = "ExampleQM";  // target queue manager
  public int      waitFor    =   10000;      // millsecs to wait for reply

/*
 * Send a completed admin message.
 *   Uses the simple putMessage method which is not assured if the
 *   the queue is defined for synchronous operation.
 */
public void sendRequest( MQeAdminMsg msg ) throws Exception
{
  myQM.putMessage( targetQMgr,
                   MQe.Admin_Queue_Name,
                   msg,
                   null,
                   0 );
}

/*
 * Wait a while for a reply message. This method will wait for
 * a limited time on either a local or a remote reply to queue.
 * Parameters:
 *   msgTest: a filter for the reply message to wait for
 * Returns:
 *   respMsg: a reply message matching the msgTest filter.
 */
public MQeAdminMsg waitForReply( MQeFields msgTest ) throws Exception
{
  MQeAdminMsg respMsg = null;
  respMsg = (MQeAdminMsg)myQM.waitForMessage(targetQMgr,
                                             MQe.Admin_Reply_Queue_Name,
                                             msgTest,
                                             null,
                                             0,
                                             waitFor);
  return respMsg;
}
```

The basic administration reply message

Once an administration request has been processed, a reply, if requested, is sent to the reply-to queue manager queue. The reply message has the same basic format as the request message with some additional fields.

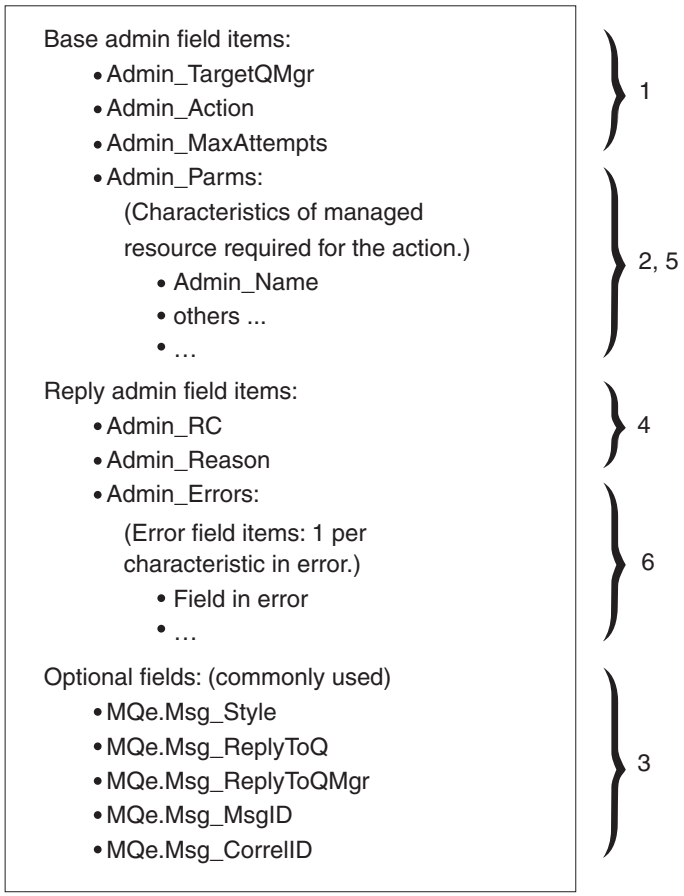

Figure 11. Administration reply message

A reply is made up of:

1. Base administration fields. These are copied from the request message
2. Administration fields that are specific to the resource being managed
3. Optional fields to assist with the processing of administration messages. These are copied from the request message
4. Administration fields detailing outcome of request
5. Administration fields providing detailed results of the request that are specific to the resource being managed
6. Administration fields detailing errors that are specific to the resource being managed

The first three items are describe in "The basic administration request message" on page 84. The reply specific fields are described in the following sections.

Outcome of request fields

Admin_RC field

This integer field contains the overall outcome of the request. This is a field of type `int` that is set to one of:

MQeAdminMsg.RC_Success
The action completed successfully

MQeAdminMsg.RC_Failed
The request failed completely.

MQeAdminMsg.RC_Mixed
The request was partially successful. A mixed return code could result if a request is made to update four attributes of a queue and three succeed and one fails.

is available in field that is of type `Unicode`.

Admin_Reason
A unicode field containing the overall reason for the failure in the case of `Mixed` and `Failed`.

Admin_Parms
An MQeFields object contains a field for each characteristics of the managed resource

Admin_Errors
An MQeFields object containing one field for each update that failed. Each entry contained in the *Admin_Errors* field is of type ascii or asciiArray.

The following methods are available for getting some of the reply fields:

Table 7. Getting administration reply fields

Adminisration field	Field type	get method
Admin_RC	int	int getAction()
Admin_Reason	unicode	String getReason()
Admin_Parms	MQeFields	MQeFields getOutputFields()
Admin_Errors	MQeFields	MQeFields getErrorFields()

Depending on the action performed, the only fields of interest may be the return code and reason. This is the case for **delete**. For other actions such as **inquire**, more details may be required in the reply message. For instance, if an **inquire** request is made for fields *Queue_Description* and *Queue_FileDesc*, the resultant MQeFields object would contain the values for the actual queue in these two fields.

The following table shows the *Admin_Parms* fields of a request message and a reply message for an inquire on several parameters of a queue:

Table 8. Enquiring on queue parameters

Admin_Parms field fame	Request message		Reply message	
	Type	Value	Type	Value
Admin_Name	ascii	″TestQ″	ascii	″TestQ″
Queue_QMgrName	ascii	″ExampleQM″	ascii	″ExampleQM″
Queue_Description	Unicode	null	Unicode	″A test queue″
Queue_FileDesc	ascii	null	ascii	″c:\queues\″

For actions where no additional data is expected on the reply, the *Admin_Parms* field in the reply matches that of the request message. This is the case for the **create** and **update** actions.

administration reply message

Some actions, such as **create** and **update**, may request that several characteristic of a managed resource be set or updated. In this case, it is possible for a return code of `RC_Mixed` to be received. Additional details indicating why each update failed are available from the *`Admin_Errors`* field. The following table shows an example of the *`Admin_Parms`* field for a request to update a queue and the resultant *`Admin_Errors`* field:

Table 9. Request and reply message to update a queue

Field name	Request message		Reply message	
	Type	**Value**	**Type**	**Value**
Admin_Parms field				
Admin_Name	ascii	″TestQ″	ascii	″TestQ
Queue_QMgrName	ascii	″ExampleQM″	ascii	″ExampleQM″
Queue_Description	Unicode	null	Unicode	″ExampleQM″ ″A new description″
Queue_FileDesc	ascii	null	Unicode	″D:\queues″
Admin_Errors field				
Queue_FileDesc	n/a	n/a	ascii	″Code=4;com.ibm.mqe.MQeException: wrong field type″

For fields where the update or set is successful there is no entry in the *`Admin_Errors`* field.

A detailed description of each error is returned in an ascii string. The value of the error string is the exception that occurred when the set or update was attempted. If the exception was an MQeException, the actual exception code is returned along with the *`toString`* representation of the exception. So, for an MQeException, the format of the value is:

```
"Code=nnnn;toString representation of the exception"
```

The following code fragment shows how to check the outcome of an administration request and to send any errors to System.out.

```
/**
 * Check to see if a good reply was received.
 * If not detail the error(s) that occurred
 * @return boolean  true if good
 * @param replyMsg reply message to check
 * Throws an Exception if the request failed.
 */
public boolean checkReply( MQeAdminMsg replyMsg ) throws Exception
{
  // Was a reply received ?
  if (replyMsg == null)
  {
    System.out.println("..No response received to the request");
    throw new Exception("No response message received");
  }
  // If the reply was not successful output details for failure
  if ( replyMsg.getRC() != MQeAdminMsg.RC_Success)
  {
    System.out.println("..Action Failed: "+replyMsg.getReason());

    // If mixed then detail each error that occurred
    if ( replyMsg.getRC() == MQeAdminMsg.RC_Mixed)
    {
```

```
      MQeFields errors = replyMsg.getErrorFields();
      Enumeration en = errors.fields();
      // process each error
      while( en.hasMoreElements() )
      {
        String value[];
        String name = (String)en.nextElement();
        // Field in error may be an array
        if ( errors.dataType( name ) == MQeField.TypeArrayElements )
          value = errors.getAsciiArray( name );
        else
          value = new String[] { errors.getAscii( name ) };
        for (int j=0; j<value.length; j++)
          System.out.println("Field in error: "+name+"  "+value[j]);
      }
    }
    // Request failed so throw exception
    throw new MQeException(replyMsg.getReason());
  }
  return true;    // All is OK
}
```

Administration of managed resources

As described in previous sections, MQSeries Everyplace has a set of resources that can be administered with administration messages. These resources are known as *managed resources*. The following sections provide information on how to manage some of these resources. For detailed description of the application programming interface for each resource see the *MQSeries Everyplace for Multiplatforms, Programming Reference*.

Queue managers

The complete management life-cycle for most managed resources can be controlled with administration messages. This means that the managed resource can be brought into existence, managed and then deleted with administration messages. This is not the case for queue managers. Before a queue manager can be managed it must be created and started. See “Creating and deleting queue managers” on page 31 for information on creating and starting a queue manager.

The queue manager has very few characteristics itself, but it controls other MQSeries Everyplace resources. When you inquire on a queue manager, you can obtain a list of connections to other queue managers and a list of queues that the queue manager can work with. Each list item is the name of either a connection or a queue. Once you know the name of a resource, you can use the appropriate message to manage the resource. For instance you use an MQeConnectionAdmin message to for manage connections.

Connections

Connections define how to connect one queue manager to another queue manager. Once a connection has been defined, it is possible for a queue manager to put messages to queues on the remote queue manager. The following diagram shows the constituent parts that are required for a remote queue on one queue manager to communicate with a queue on a different queue manager:

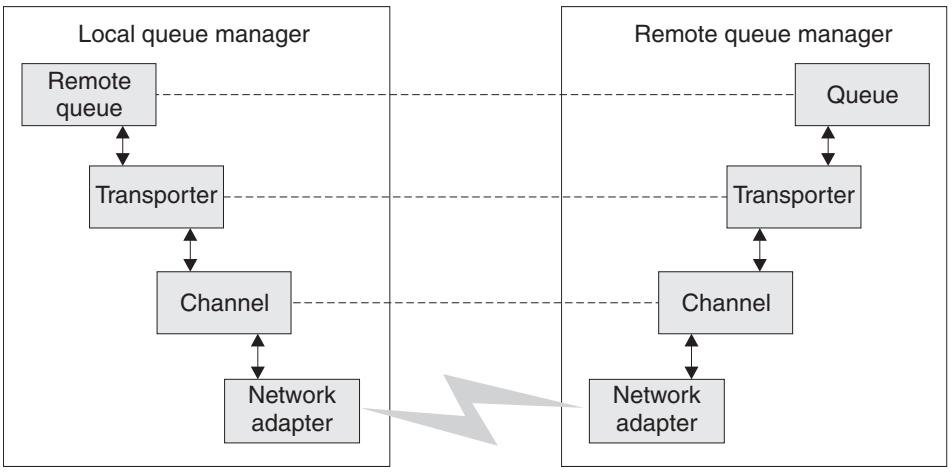

Figure 12. Queue manager connections

Communication happens at different levels:

Transporter:
Logical connection between two queues

Channel:
Logical connection between two systems

Adapter:
Protocol specific communication

The channel and adapter are specified as part of a connection definition. The transporter is specified as part of a remote queue definition. The following example code shows a method that instantiates and primes an MQeConnectionAdminMsg ready to create a connection:

```
/**
 * Setup an admin msg to create a connection definition
 */
public MQeConnectionAdminMsg addConnection( remoteQMgr
  adapter,
               parms,
               options,
               channel,
               desc ) throws Exception
{
  String remoteQMgr = "ServerQM";
  /*
   * Create an empty queue manager admin message and parameters field
   */
  MQeConnectionAdminMsg msg = new MQeConnectionAdminMsg();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of queue manager to add routes to
   */
  msg.setName( remoteQMgr );

  /*
   * Set the admin action to create a new queue
   * The connection is setup to use a default channel. This is an alias
```

```
   * which must have be setup on the queue manager for the connection to
   * work.
   */
  msg.create( adapter,
              parms,
              options,
              channel,
              desc );

  return msg;
}
```

MQSeries Everyplace provides a choice of channel and adapter types. Depending on the selection, queue managers can be connected in the following ways:

- Client to server
- Peer to peer

Client to server

In a client server configuration, one queue manager acts as a client and the other runs in a server environment. A server allows multiple simultaneous incoming connections (channels). To accomplish this the server must have components that can handle multiple incoming requests. See "Server" on page 42 for a description of how to run a queue manager in a server environment.

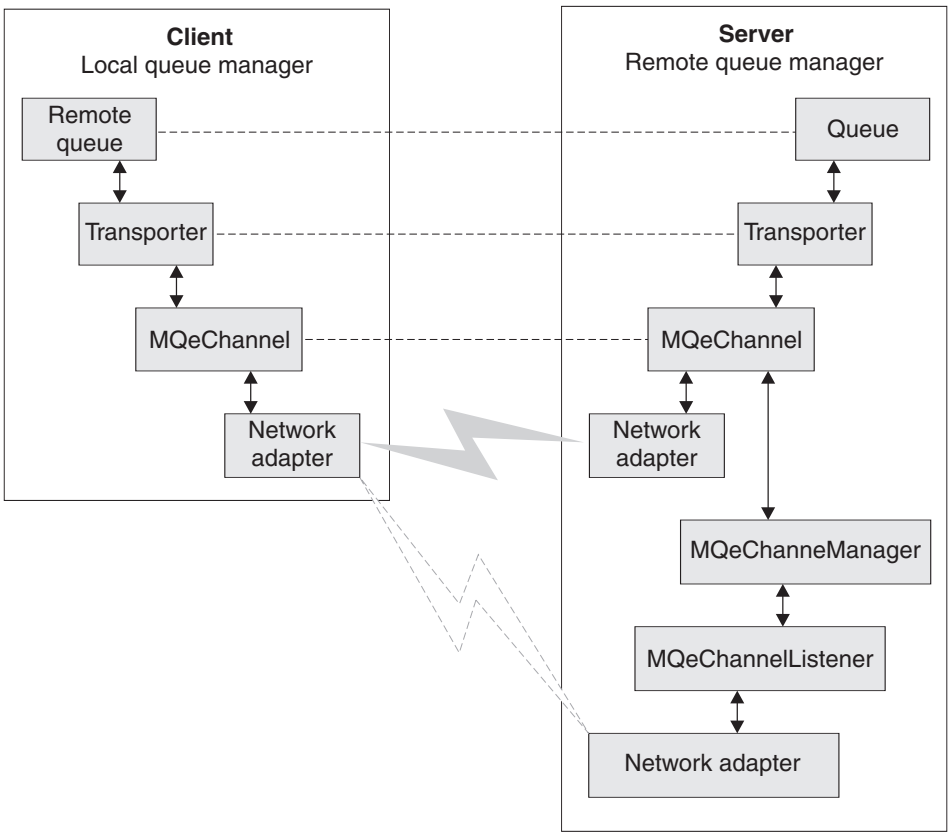

Figure 13. Client to server connections

Figure 13 shows the typical connection components in a client server configuration.

You use MQeConnectionAdminMsg to configure the client portion of a connection. The channel type is `com.ibm.mqe.MQeChannel`. Normally an alias of `DefaultChannel`

is configured for MQeChannel. The following code fragment shows how to configure a connection on a client to communicate with a server using the http protocol.

```
/**
 * Create a connection admin message that creates a connection
 * definition to a remote queue manager using the HTTP protocol. Then
 * send the message to the client queue manager.
 */
public addClientConnection( MQeQueueManager myQM,
  String targetQMgr ) throws Exception
{
  String remoteQMgr  = "ServerQM";
  String adapter     = "Network:127.0.0.1:80";
// This assumes that an alias called Network has been setup for
// network adapter com.ibm.mqe.adapters.MQeTcpipHttpAdapter
  String parameters  = null;
  String options     = null;
  String channel     = "DefaultChannel";
  String description = "client connection to ServerQM";

  /*
   * Setup the admin msg
   */
  MQeConnectionAdminMsg msg = addConnection( remoteQMgr,
                                             adapter,
                                             parameters,
                                             options,
                                             channel,
                                             desc );

  /*
   *  Put the admin message to the admin queue (not using assured flows)
   */
  myQM.putMessage(targetQMgr,
  MQe.Admin_Queue_Name,
  msg,
  null,
  0 );

}
```

Peer to peer

In a peer to peer configuration, a queue manager running as a peer can talk to many other peers simultaneously but can only have one other peer talk to it at any time. One peer is configured as a master or initiator, the other as a slave or receiver.

You configure the master in much the same way as a client connection definition, the only difference being the type of channel to use. The channel type must be set to `com.ibm.mqe.adapters.MQePeerChannel` (or an alias).

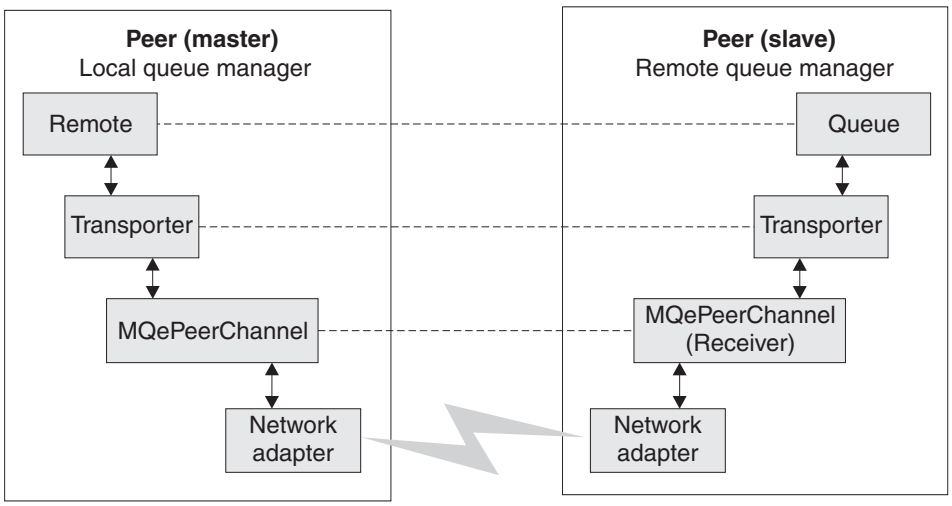

Figure 14. Peer to peer connections

You configure the slave or receiver in a similar way but with the following differences:

- The connection definition must have the same name as the queue manager it is defined on
- The channel type must be `com.ibm.mqe.adapters.MQePeerChannel`
- The adapter must be configured as a listener

The following code fragment configures a queue manager called `PeerQM1` as a peer receiver, listening on port 8081 using the HTTP protocol.

```
/**
 * Create a connection admin message which will create a connection
 * definition to a remote queue manager using the HTTP protocol. Then
 * send the message to the client queue manager.
 */
public addClientConnection( MQeQueueManager myQM,
  String targetQMgr ) throws Exception
{
  String remoteQMgr  = "PeerQM1";
// To be a receiver the connection definition called "PeerQM1" must
// be configured on queue manager "PeerQM1"
  String adapter     = "Network::8081";
// This assumes that an alias called Network has been setup for
// network adapter com.ibm.mqe.adapters.MQeTcpipHttpAdapter
  String parameters  = null;
  String options     = null;
  String channel     = "com.ibm.mqe.adapters.MQePeerChannel";
  String description = "peer receiver on PeerQM";

  /*
   * Setup the admin msg
   */
  MQeConnectionAdminMsg msg = addConnection( remoteQMgr,
  adapter,
              parameters,
              options,
              channel,
              desc );

  /*
   *  Put the admin message to the admin queue (not using assured flows)
   */
  myQM.putMessage(targetQMgr,
```

```
  MQe.Admin_Queue_Name,
  msg,
  null,
  0 );
}
```

The following table shows the connection definition parameters for a receiver on `PeerQM1` and for any other peer queue manager that would like to communicate with it.

Table 10. Peer-to-peer connection definitions

	Master (Initiator)	**Slave (Receiver)**
Queue Manager	Any	″PeerQM1″
Connection name	″PeerQM1″	″PeerQM1″
Channel	com.ibm.mqe.MQePeerChannel	com.ibm.mqe.MQePeerChannel
Adapter	Network:192.168.0.10:8081	Network::8081

Adapters

For details of the adapters supplied with MQSeries Everyplace see the “Chapter 10. MQSeries Everyplace adapters” on page 205 and Chapter 9 in the MQSeries Everyplace for Multiplatforms, Programming Reference.

Routing connections

You can set up a connection so that a queue manager routes messages through an intermediate queue manager. This requires two connections:

1. A connection to the intermediate queue manager
2. A connection to the target queue manager

The first connection is created by the methods described earlier in this section, either as a client or as a peer connection. For the second connection, the name of the intermediate queue manager is specified in place of the network adapter name. With this configuration an application can put messages to the target queue manager but route them through one or more intermediate queue managers.

Aliases

You can assign multiple names or aliases to a connection (see “Aliases” on page 38). When an application calls methods on the MQeQueueManager class that require a queue manager name be specified, it can also use an alias.

You can alias both local and remote queue managers. To alias a local queue manager, you must first establish a connection definition with the same name as the local queue manager. This is a logical connection that can have all parameters set to null.

To add and remove aliases use the **Action_AddAlias** and **Action_RemoveAlias** actions of the MQeConnectionAdminMsg class. You can add or remove multiple aliases in one message. Put the aliases that you want to manipulated directly into the message by setting the ascii array field *Con_Aliases*. Alternatively you can use the two methods **addAlias()** or **removeAlias()**. Each of these methods takes one alias name but you can call the method repeatedly to add multiple aliases to a message. The following snippet of code shows how to add connection aliases to a message:

```
/**
 * Setup an admin msg to add aliases to a queue manager (connection)
 */
public MQeConnectionAdminMsg addAliases(  String queueManagerName
```

```
                                        String aliases[] ) throws Exception
{
  /*
   * Create an empty connection admin message
   */
  MQeConnectionAdminMsg msg = new MQeConnectionAdminMsg();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of the connection to add aliases to
   */
  msg.setName( queueManagerName );

  /*
   * Use the addAlias method to add aliases to the message.
   */
  for ( int i=0; i<aliases.length; i++ )
  {
    msg.addAlias( aliases[i] );
  }

  return msg;
}
```

Queues

The queue types provided by MQSeries Everyplace are described briefly in "MQSeries Everyplace queues" on page 3. The simplest of these is a local queue that is implemented in class MQeQueue and is managed by class MQeQueueAdminMsg. All other types of queue inherit from MQeQueue. For each type of queue there is a corresponding administration message that inherits from MQeQueueAdminMsg. The following sections describe the administration of the various types of queues.

Local queue

You can create, update, delete and inquire on local queues and their descendents using administration actions provided in MQSeries Everyplace. The basic administration mechanism is inherited from MQeAdminMsg.

The name of a queue is formed from the target queue manager name (for a local queue this is the name of the queue manager that owns the queue) and a unique name for the queue on that queue manager. Two fields in the administration message are used to uniquely identify the queue, these are the ascii fields *Admin_Name* and *Queue_QMgrName*. You can use the **setName(queueManagerName, queueName)** method to set these two fields in the administration message.

The diagram below shows an example of a queue manager configured with a local queue. Queue manager `qm1` has a local queue named `invQ`. The queue manager name characteristic of the queue is qm1, which matches the queue manager name.

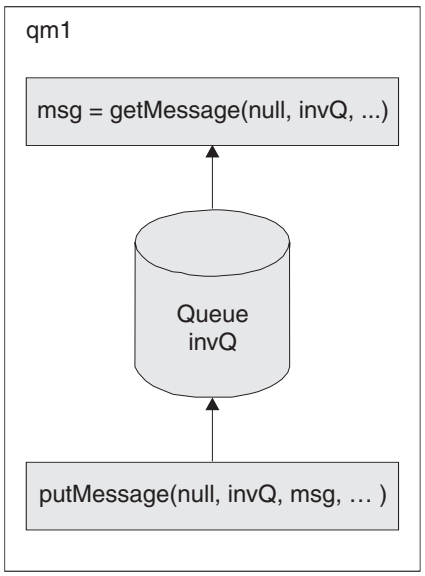

Figure 15. Local queue

Message Store: Local queues require a message store to store their messages. Each queue can specify what type of store to use, and where it is located. Use the queue characteristic *Queue_FileDesc* to specify the type of message store and to provide parameters for it. The field type is `ascii` and the value must be a file descriptor of the form:

adapter class:adapter parameters
or
adapter alias:adapter parameters

For example:

```
MsgLog:d:\QueueManager\ServerQM12\Queues
```

MQSeries Everyplace Version 1.1 provides two adapters, one for writing messages to disk and one for storing them in memory. By creating an appropriate adapter, messages can be stored in any suitable place or medium (such as DB2 data base or writable CDs).

The choice of adapter determines the persistence and resilience of messages. For instance if a memory adapter is used then the messages are only as resilient as the memory. Memory may be a much faster medium than disk but is highly volatile compared to disk. Hence the choice of adapter is an important one.

If you do not provide message store information when creating a queue, it defaults to the message store that was specified when the queue manager was created. See “Chapter 4. Queue managers, messages, and queues” on page 31 for more details.

The following should be taken into consideration when setting the *Queue_FileDesc* field:

- Ensure that the correct syntax is used for the system that the queue resides on. For instance, on a windows system use "\" as a file separator on UNIX® systems use "/" as a file separator. In some cases it may be possible to use either but this is dependent on the support provided by the JVM (Java Virtual Machine) that the queue manager runs in. As well as file separator differences, some systems use drive letters like Windows NT whereas others like UNIX do not.

- On some systems it is possible to specify relative directories (".\") on others it is not. Even on those where relative directories can be specified, they should be used with great caution as the current directory can be changed during the lifetime of the JVM. Such a change causes problems when interacting with queues using relative directories.

Creating a local queue: The following code fragment demonstrates how to create a local queue:

```
/**
 * Create a new local queue
 */
protected void createQueue(MQeQueueManager localQM,
                           String          qMgrName,
                           String          queueName,
                           String          description,
                           String          queueStore
  ) throws Exception
{
  /*
   * Create an empty queue admin message and parameters field
   */
  MQeQueueAdminMsg msg = new MQeQueueAdminMsg();
  MQeFields parms = new MQeFields();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of queue to manage
   */
  msg.setName( qMgrName, queueName );

  /*
   * Add any characteristics of queue here, otherwise
   * charateristics will be left to default values.
   /
  if ( description != null ) // set the description ?
    parms.putUnicode( MQeQueueAdminMsg.Queue_Description,
                      description);

  if ( queueStore != null )  // Set the queue store ?
    // If queue store includes directory and file info then it
    // must be set to the correct style for the system that the
    // queue will reside on e.g \ or /
    parms.putAscii(MQeQueueAdminMsg.Queue_FileDesc,
                   queueStore );
  /*
   * Other queue characteristics like queue depth, message expiry
   * can be set here à
   */

  /*
   * Set the admin action to create a new queue
   */
  msg.create( parms );

  /*
   *  Put the admin message to the admin queue (not assured delivery)
   */
  localQM.putMessage( qMgrName,
                      MQe.Admin_Queue_Name,
                      msg,
```

```
                    null,
                    0);
}
```

Queue security: Access and security are owned by the queue and may be granted for use by a remote queue manager (when connected to a network), allowing these other queue managers to send or receive messages to the queue. The following characteristics are used in setting up queue security:

- *Queue_Cryptor*
- *Queue_Authenticator*
- *Queue_Compressor*
- *Queue_TargetRegistry*
- *Queue_AttrRule*

For more detailed information on setting up queue based security see "Chapter 8. Security" on page 157.

Other queue characteristics: You can configure queues with many other characteristics such as the maximum number of messages that are permitted on the queue. For a description of these, see the *MQeQueueAdminMsg* section of the *MQSeries Everyplace for Multiplatforms, Programming Reference*.

Aliases: Queue names can have aliases similar to those described for connections in "Aliases" on page 98. The code fragment in the connections section alias example shows how to setup aliases on a connection, setting up aliases on a queue is the same except that the an MQeQueueAdminMsg is used instead of an MQeConnectionAdminMsg.

Action restrictions: Certain administrative actions can only be performed when the queue is in a predefined state, as follows:

Action_Update

- If the queue is in use, characteristics of the queue cannot be changed
- The security characteristics of the queue cannot be changed if there are messages on the queue
- The queue message store cannot be changed once it has been set

Action_Delete

The queue cannot be deleted if the queue is in use or if there are messages on the queue

If the request requires that the queue is not in use, or that it has zero messages, the administration request can be retried, either when the queue manager restarts or at regular time intervals. See "The basic administration request message" on page 84 for details on setting up an administration request retry.

Remote queue

Remote queues are implemented by the MQeRemoteQueue class and are managed with the MQeRemoteQueueAdminMsg class which is a subclass of MQeAdminMsg.

The name of a queue is formed from the target queue manager name (for a remote queue this is the name of the queue manager where the queue is local) and the real name of the queue on that queue manager. Two fields in the administration message are used to uniquely identify the queue, these are the ascii fields *Admin_Name* and *Queue_QMgrName*. You can use the **setName(queueManagerName,**

queueName) method to set these two fields in the administration message. For a remote queue definition, the queue manager name of the queue never matches the name of the queue manager where the definition resides.

The remote definition of the queue should, in most cases, match that of the real queue. If this is not the case different results may be seen when interacting with the queue. For instance:

- For asynchronous queues if *max message size* on the remote definition is greater than that on the real queue, the message is accepted for storage on the remote queue but may be rejected when moved to the real queue. The message is not lost, it remains on the remote queue but cannot be delivered.
- If the security characteristics for a synchronous queue do not match, MQSeries Everyplace negotiates with the real queue to decide what security characteristics should be used. In some cases the message put is successful, in others an attribute mismatch exception is returned.

Setting the operation mode: To set a queue for synchronous operation, set the *Queue_Mode* field to `Queue_Synchronous`.

Asynchronous queues require a message store to temporarily store messages. Definition of this message store is the same as for local queues (see "Message Store" on page 100).

To set a queue for asynchronous operation, set the *Queue_Mode* field to `Queue_Asynchronous`.

Figure 16 on page 104 shows an example of a remote queue set up for synchronous operation and a remote queue setup for asynchronous operation.

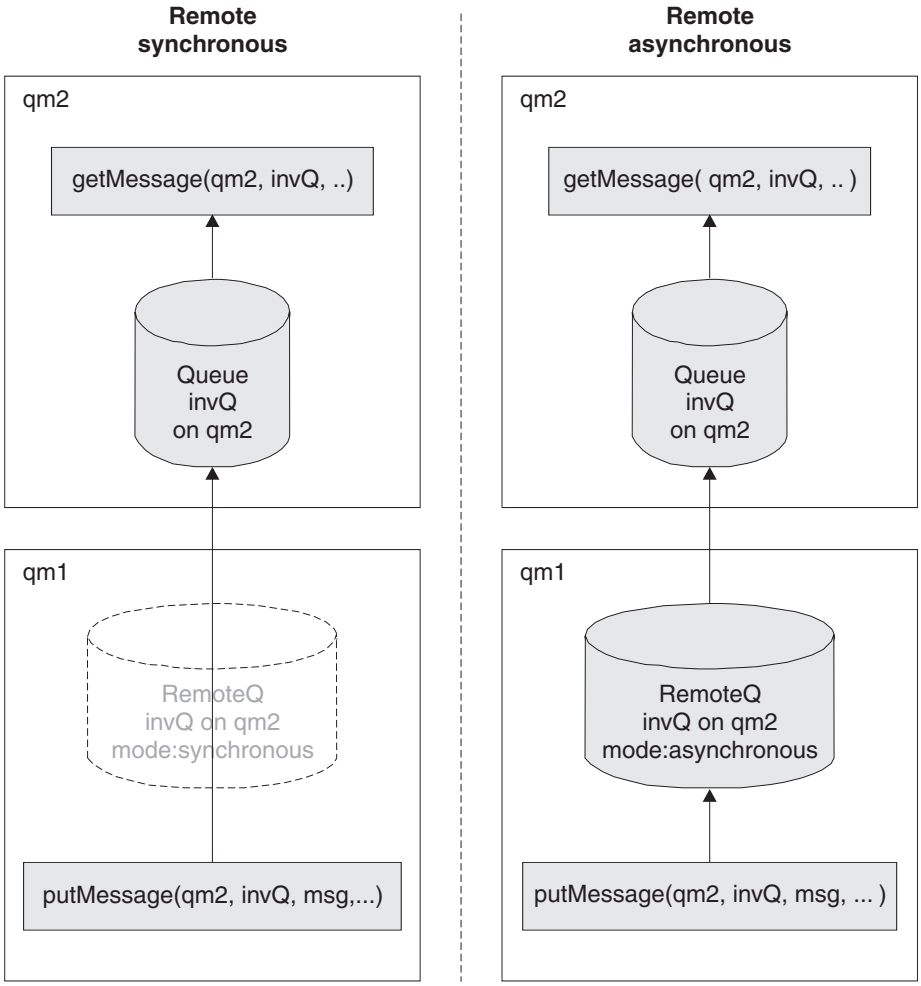

Figure 16. Remote queue

- In both the synchronous and asynchronous examples queue manager `qm2` has a local queue `qm2`
- In the synchronous example, queue manager `qm1` has a remote queue definition of queue `invQ`. `invQ` resides on queue manager `qm2`. The mode of operation is set to synchronous.

  An application using queue manager `qm1` and putting messages to queue `qm2.invQ` establishes a network connection to queue manager `qm2` (if it does not already exist) and the message is immediately put on the real queue. If the network connection cannot be established then the application receives an exception that it must handle.
- In the asynchronous example, queue manager `qm1` has a remote queue definition of queue `invQ`. `invQ` resides on queue manager `qm2`. The mode of operation is set to asynchronous.

  An application using queue manager `qm1` and putting messages to queue `qm2.invQ` stores messages temporarily on the remote queue on `qm1`. When the transmission rules allow, the message is moved to the real queue on queue manager `qm2`. The message remains on the remote queue until the transmission is successful.

Creating a remote queue: The following code fragment shows how to setup an administration message to create a remote queue.

```
/**
 * Create a remote queue
 */
protected void createQueue(MQeQueueManager localQM,
                           String          targetQMgr,
                           String          qMgrName,
                           String          queueName,
       String          description,
                           String          queueStore,
       byte            queueMode
  ) throws Exception
{
  /*
   * Create an empty queue admin message and parameters field
   */
  MQeRemoteQueueAdminMsg msg = new MQeRemoteQueueAdminMsg();
  MQeFields parms = new MQeFields();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of queue to manage
   */
  msg.setName( qMgrName, queueName );

  /*
   * Add any characteristics of queue here, otherwise
   * charateristics will be left to default values.
   /
  if ( description != null ) // set the description ?
    parms.putUnicode( MQeQueueAdminMsg.Queue_Description,
                      description);

  // set the queue access mode if mode is valid
  if ( queueStore != MQeQueueAdminMsg.Queue_Asynchronous &&
       queueStore != MQeQueueAdminMsg.Queue_Synchronous )
    throw new Exception ("Invalid queue store");

  parms.putByte( MQeQueueAdminMsg.Queue_Mode,
                 queueMode);

  if ( queueStore != null )  // Set the queue store ?
    // If queue store includes directory and file info then it
    // must be set to the correct style for the system that the
    // queue will reside on e.g \ or /
    parms.putAscii( MQeQueueAdminMsg.Queue_FileDesc,
                    queueStore );
  /*
   * Other queue characteristics like queue depth, message expiry
   * can be set here à
   */

  /*
   * Set the admin action to create a new queue
   */
  msg.create( parms );

  /*
   *  Put the admin message to the admin queue (not assured delivery)
   *  on the target queue manager
   */
  localQM.putMessage( targetQMgr,
                      MQe.Admin_Queue_Name,
```

```
                    msg,
                    null,
                    0);
}
```

For synchronous operation, the queue characteristics for inclusion in the remote queue definition can be obtained using *queue discovery* which is explained on page 63.

Store-and-forward queue

This type of queue is normally defined on a server and can be configured in the following ways:

- Forward messages to the next queue manager. The next queue manager may not be the target queue manager. In this case the store-and-forward queue *pushes* messages to the next hop.
- Hold messages until the target queue manager can collect the messages from the store-and-forward queue. This can be accomplished using a *home-server* queue (see "Home-server queue" on page 109). Using this approach messages are *pulled* from the store-and-forward queue.

Store-and-forward queues are implemented by the MQeStoreAndForwardQueue class. They are managed with the MQeStoreAndForwardQueueAdminMsg class, which is a subclass of MQeRemoteQueueAdminMsg. The main addition in the subclass is the ability to add and remove the names of queue managers for which the store-and-forward queue can hold messages. You can add and delete queue manager names with the **Action_AddQueueManager** and **Action_RemoveQueueManager** actions. You can add or remove multiple queue manager names with one administration message. You can put the names directly into the message by setting the ascii array field *Queue_QMgrNameList*. Alternatively you can use the **addQueueManager()** and **removeQueueManager()** methods. Each of these methods takes one queue manager name but you can call the method repeatedly to add multiple queue managers to a message.

The following code fragment shows how to add target queue manager names to a message:

```
/**
 * Setup an admin msg to add target queue managers to
 * a store and forward queue.
 */
public MQeStoreAndForwardQueueAdminMsg addQueueManager( String queueName
                                                         String queueManagerName
                                                         String qMgrNames[] )
                                                         throws Exception
{
  /*
   * Create an empty admin message
   */
  MQeStoreAndForwardQueueAdminMsg msg =
  new MQeStoreAndForwardQueueAdminMsg();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of the store and forward queue
   */
  msg.setName( queueManagerName, queueName );
```

```
  /*
   * Use the addAlias method to add aliases to the message.
   */
  for ( int i=0; i<qMgrNames.length; i++ )
  {
    msg.addQueueManager(qMgrNames[i] );
  }

  return msg;
}
```

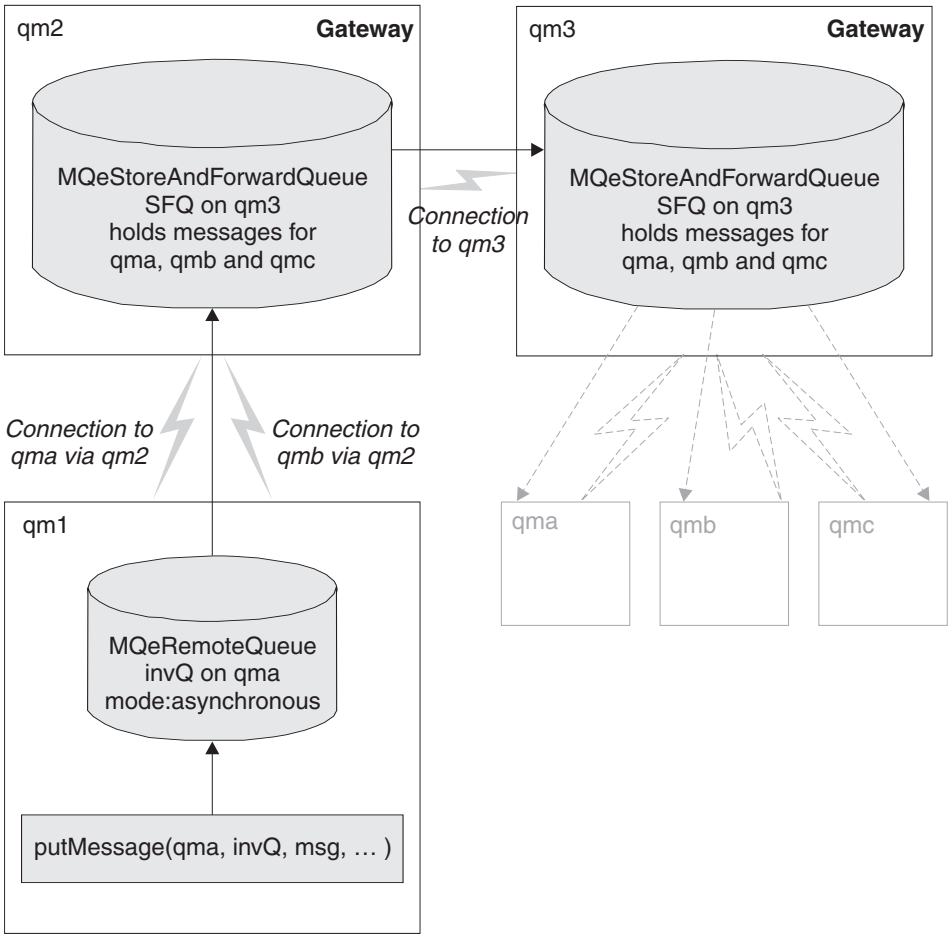

Figure 17. Store-and-forward queue

If you want the store-and-forward queue to push messages to the next queue manager, the queue manager name attribute of the store-and-forward queue must be the name of the next queue manager. You must also configure a connection to the next queue manager.

If you want the store-and-forward queue to wait for messages to be collected (pulled), the queue manager name attribute of the store-and-forward queue has no meaning. The only restriction on the queue manager attribute of the queue name is that there must not be a connection with the same name. If there is such a connection, the queue tries use the connection to forward messages.

Figure 17 on page 107 shows an example of two store and forward queues on different queue managers, one setup to push messages to the next queue manager, the other setup to wait for messages to be collected:

- Queue manager qm2 has a connection configured to queue manager qm3
- Queue manager qm2 has a store-and-forward queue configuration that pushes messages using connection qm3, to queue manager qm3. Note that the queue manager name portion of the store-and-forward queue is qm3 which matches the connection name
- Store-and-forward queue qm3.SFQ on qm2 has been configured to handle messages that are destined for queue managers qma, qmb and qmc.
- Queue manager qm3 has a store-and-forward queue qm3.SFQ. The queue manager name portion of the queue name qm3 does not have a corresponding connection called qm3, so all messages are stored on the queue until they are collected.
- Store-and-forward queue qm3.SFQ on qm3 holds messages on behalf of queue managers qma, qmb and qmc. Messages are stored until they are collected or they expire.

If a queue manager wants to send a message to another queue manager using a store-and-forward queue on an intermediate queue manager, the initiating queue manager must have:

- A connection configured to the intermediate queue manager
- A connection configured to the target queue manager routed through the intermediate queue manager
- A remote queue definition for the target queue

When these conditions are fulfilled, an application can put a message to the target queue on the target queue manager without having any knowledge of the layout of the queue manager network. This means that changes to the underlying queue manager network do not affect application programs.

In Figure 17 on page 107 queue manager qm1 has been configured to allow messages to be put to queue invQ on queue manager qma. The configuration consists of:

- A connection to the intermediate queue manager qm2
- A connection to the target queue manager qma
- A remote asynchronous queue invQ on qma

If an application program uses queue manager qm1 to put a message to queue invQ on queue manager qma the message flows as follows:

1. The application puts the message to asynchronous queue qma.invQ. The message is stored locally on qm1 until transmission rules allow the message to be moved to the next hop
2. When transmission rules allow, the message is moved. Based on the connection definition for qma, the message is routed to queue manager qm2
3. The only queue configured to handle messages for queue invQ on queue manager qma is store-and-forward queue qm3.SFQ on qm2. The message is temporarily stored in this queue
4. The stored and forward queue has a connection that allows it to push messages to its next hop which is queue manager qm3
5. Queue manager qm3 has a store-and-forward queue qm3.SFQ that can hold messages destined for queue manager qma so the message is stored on that queue

6. Messages for qma remain on the store-and-forward queue until they are collected by queue manager qma. See "Home-server queue" for how to set this up.

Home-server queue

Home-server queues are implemented by the MQeHomeServerQueue class. They are managed with the MQeHomeServerQueueAdminMsg class which is a subclass of MQeRemoteQueueAdminMsg. The only addition in the subclass is the *Queue_QTimerInterval* characteristic. This field is of type int and is set to a millisecond timer interval. If you set this field to a value greater than zero, the home-server queue checks the home server every n milliseconds to see if there are any messages waiting for collection. Any messages that are waiting are delivered to the target queue.

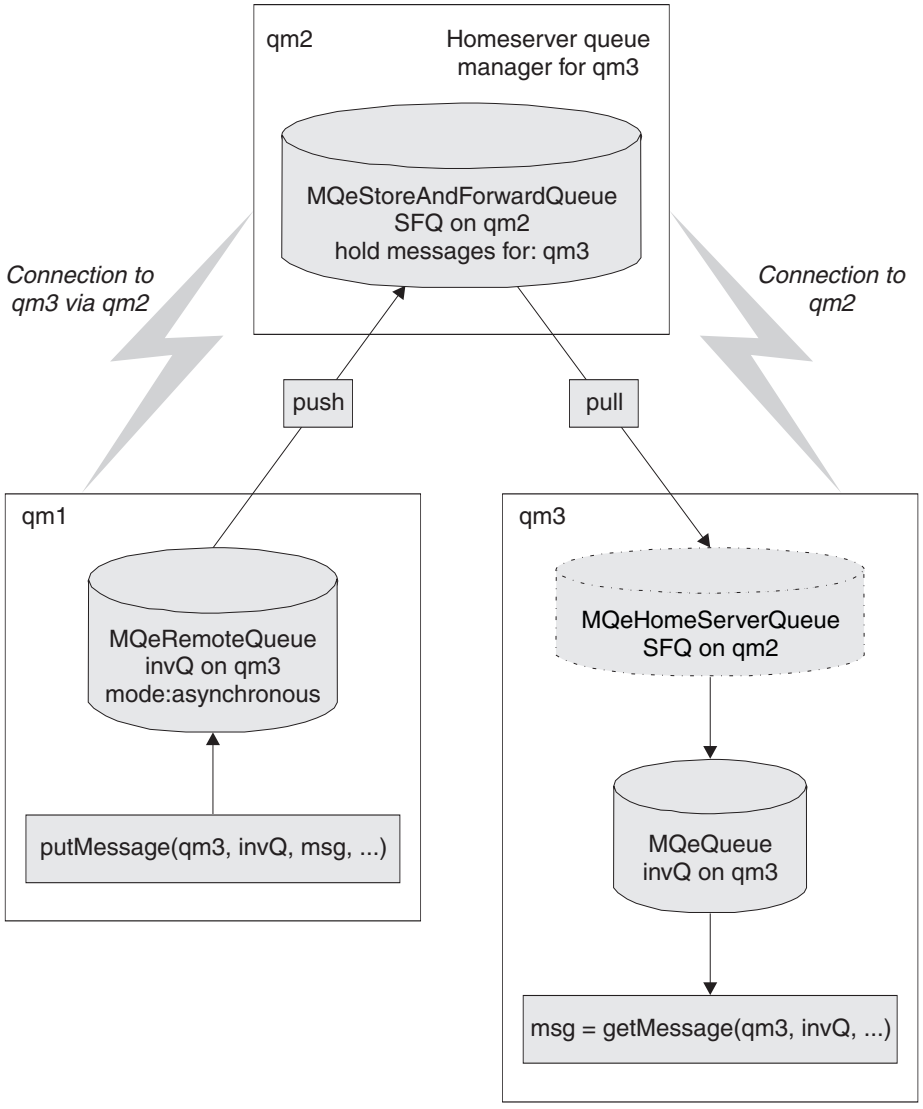

Figure 18. Home-server queue

The name of the home-server queue is set as follows:

- The queue name must match the name of the store-and-forward queue
- The queue manager attribute of the queue name must be the name of the home-server queue manager

The queue manager where the home-server queue resides must have a connection configured to the home-server queue manager.

Figure 18 on page 109 shows an example of a queue manager qm3 that has a home-server queue SFQ configured to collect messages from its home-server queue manager qm2.

The configuration consists of:

- A home server queue manager qm2
- A store and forward queue SFQ on queue manager qm2 that holds messages for queue manager qm3
- A queue manager qm3 that normally runs disconnected and cannot accept connections from queue manager qm2
- Queue manager qm3 has a connection configured to qm2
- A home server queue SFQ that uses queue manager qm2 as its home server

Any messages that are directed to queue manager qm3 through qm2 are stored on the store-and-forward queue SFQ on qm2 until the home-server queue on qm3 collects them.

MQSeries-bridge queue

An MQSeries-bridge queue is a remote queue definition that refers to a queue residing on an MQSeries queue manager. The queue holding the messages resides on the MQSeries queue manager, not on the local queue manager.

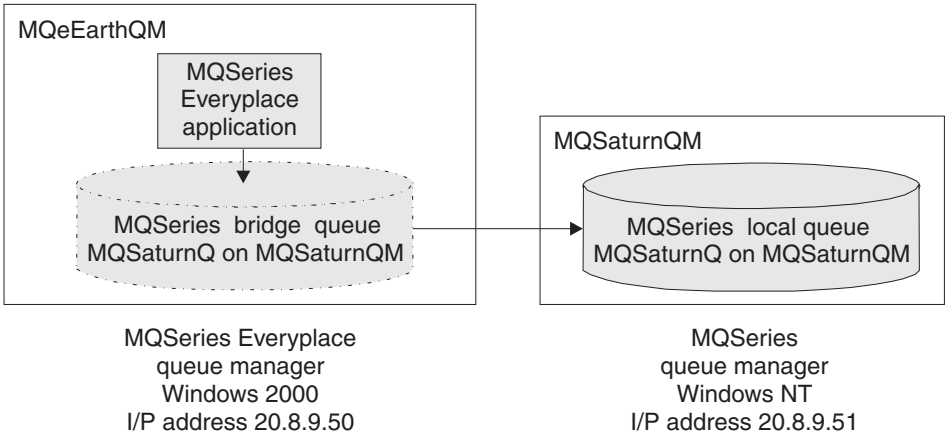

Figure 19. MQSeries-bridge queue

- The MQSaturnQM MQSeries queue manager has a local queue MQSaturnQ defined .
- The MQeEarthQM must have an MQSeries-bridge queue defined called MQSaturnQ on the MQSaturnQM queue manager.
- Applications attached to the MQeEarthQM queue manager put messages to the MQSaturnQ MQSeries-bridge queue, and the bridge queue delivers the message to the MQSaturnQ on the MQSaturnQM queue manager.

The definition of the bridge queue requires that bridge, MQSeries queue manager proxy, and client connection, names are specified to uniquely identify a client connection object in the bridge object hierarchy (see Figure 24 on page 120). This information identifies how the MQSeries-bridge accesses the MQSeries queue manager, to manipulate an MQSeries queue.

The MQSeries-bridge queue provides the facility to put to a queue on a queue manager that is not directly connected to the MQSeries-bridge. This allows a message to be sent to an MQSeries queue manager (the target) routed through another MQSeries queue manager. The MQSeries-bridge queue takes the name of the target queue manager and the intermediate queue manager is named by the MQSeries queue manager proxy.

For a complete list of the characteristics used by the MQSeries-bridge queue, see *MQeMQBridgeQueueAdminMsg* in the *com.ibm.mqe.bridge* section of*MQSeries Everyplace for Multiplatforms, Programming Reference*.

Table 11 details the list of operations supported by the MQSeries-bridge queue, once it has been configured:

Table 11. Message operations supported by MQSeries—bridge queue

Type of operation	Supported by MQSeries-bridge queue
getMessage()	no
putMessage()	yes
browseMessage()	no
browseAndLockMessage	no

If an application attempts to use one of the unsupported operations, an MQeException of `Except_NotSupported` is returned.

When an application puts a message to the bridge queue, the bridge queue takes a logical connection to the MQSeries queue manager from the pool of connections maintained by the bridge's client connection object. The logical connection to MQSeries is supplied by either the MQSeries Java Bindings classes, or the MQSeries Java Client classes. The choice of classes depends on the value of the hostname field in the MQSeries queue manager proxy settings. Once the MQSeries-bridge queue has a connection to the MQSeries queue manager, it attempts to put the message to the MQSeries queue.

An MQSeries-bridge queue must always have an access mode of synchronous and cannot be configured as an asynchronous queue. This means that, if your put operation is directly manipulating an MQSeries-bridge queue and returns success, your message has passed to the MQSeries system while your process was waiting for the put operation to complete.

If you do not wish to use synchronous operations against the MQSeries-bridge queue, you may set up an asynchronous remote queue definition (see "Asynchronous messaging" on page 62) that refers to the MQSeries-bridge queue. Alternatively you can set up a store-and-forward queue, and home-server queue. These two alternative configurations provide the application with an asynchronous queue to which it can put messages. With these configurations, when your **putMessage()** method returns, the message may not necessarily have passed to the MQSeries queue manager.

An example of MQSeries-bridge queue usage is described in "Configuration example" on page 123.

Administration queue

The administration queue is implemented in class MQeAdminQueue and is a subclass of MQeQueue so it has the same features as a local queue. It is managed using administration class MQeAdminQueueAdminMsg.

If a message fails because the resource to be administered is in use, it is possible to request that the message be retried. "The basic administration request message" on page 84 provides details on setting up the maximum number attempts count. If the message fails due to the managed resource not being available and the maximum number of attempts has not been reached, the message is left on the queue for processing at a later date. If the maximum number of attempts has been reached, the request fails. By default the message is retried the next time the queue manager is started. Alternatively a timer can be set on the queue that processes messages on the queue at specified intervals. The timer interval is specified by setting the long field *Queue_QTimerInterval* field in the administration message. The interval value is specified in milliseconds.

Security and administration

By default, any MQSeries Everyplace application can administer managed resources. The application can be running as a local application to the queue manager that is being managed, or it can be running on a different queue manager. It is important that the administration actions are secure, otherwise there is potential for the system to be misused. MQSeries Everyplace provides the basic facilities for securing administration using queue-based security which is described in "Chapter 8. Security" on page 157.

If you use synchronous security, you can secure the administration queue by setting security characteristics on the queue. For example you can set an authenticator so that the user must be authenticated to the operating system (Windows NT or UNIX) before they can perform administration actions. This can be extended so that only a specific user can perform administration.

The administration queue does not allow applications direct access to messages on the queue, the messages are processed internally. This means that messages put to the queue that have been secured with message level security cannot be unwrapped using the normal mechanism of providing an attribute on a get or browse request. However, a queue rule class can be applied to the administration queue to unwrap any secured messages so that they can be processed by the administration queue. The queue rule **browseMessage()** must be coded to perform this unwrap and allow administration to take place.

Additional information on implementing queue rules can be found in "Queue rules" on page 79.

Example administration console

One of the examples provided with MQSeries Everyplace is an administration graphical user interface (GUI). This example uses many of the administration techniques and features described in previous sections of this manual. All the classes for this example are contained in package examples.administration.console.

This example demonstrates the following MQSeries Everyplace administration features:

- Management of both local and remote queue managers

- Administration of all MQSeries Everyplace managed resources
- Access to all actions of each managed resource
- Use of most of the base MQeAdminMsg features
- A queue browser
- A customized version of the queue browser for the administration reply queue.

This is provided solely as a programming example, ***it is not expected to be used outside a development and test environment***. It should be noted that this example works with other examples such as trace, and the client queue manager, and it is also subclassed to provide an administration example for the MQSeries-bridge (see "The example administration GUI application" on page 129).

The main console window

To start the console use the command:

```
java examples.administration.console.Admin
```

This displays the following window:

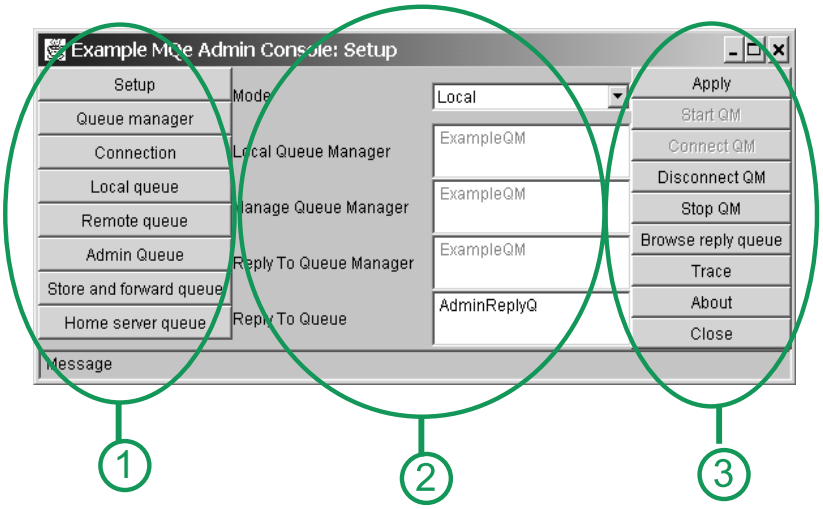

Figure 20. Administration console window

This is the central window from which all other interactions are initiated. The window has three sections:

1. **Type of resource to manage**
   The set of buttons on the left side of the window control the selection of the resource that is to be managed. There is one button for each type of MQSeries Everyplace managed resource and one special button called **Setup**. The **Setup** button provides access to a set of base administration functions such as browsing the reply-to queue and turning trace on and off.

2. **Base administration parameters**
   The central section of the window allows base administration parameters to be altered.

   Mode: Whether the queue manager to be managed is local or remote.

   Local queue manager:
   The name of the local queue manager that is initiating the

administration actions. This is set automatically when a queue manager is started with the **Start QM** button.

Remote queue manager:
If the mode is set to remote, this is the name of the queue manager to be managed. If the mode is set to local, this is always the same as the local queue manager.

Reply-to queue manager:
The name of the queue manager to which administration reply messages are to be sent.

Reply-to queue:
The name of the queue to which administration reply messages are to be sent.

3. Managed resource specific action
Each managed resource has a set of actions that can be performed on it. The buttons on the right of the main window show the actions for the resource that is selected on the left of the window. Selecting one of an action button starts the function for that action. Normally this causes the display of another window related to the action.

The selected local queue manager must be running in the JVM that the console is executing in. If it is not already running, it needs to be started using the **Start QM** button. This displays a dialog that requests the name and path of the ini file that contains the queue manager startup parameters. If the queue manager is already running, the **Connect QM** button can be selected (this is the case if administration is started from the example server ExampleAwtMQeServer).

Once the queue manager has been started, any of the resources in area 1 can be selected and managed.

Queue browser

An example queue browser, AdminQueueBrowser is provided with MQSeries Everyplace. This example shows how to browse a queue and how to display the contents of messages on the queue. The example can only browse queues that can be accessed synchronously and that the user has the necessary authority to access. The example code is not able to show the messages that are secured using message level security, .

AdminQueueBrowser has been subclassed to provide a queue browser with enhanced function for browsing the administration reply-to queue. This is implemented in class AdminLogBrowser. This subclass can be accessed by selecting the **Setup** button followed by the **Browse reply queue** button.

The following figure shows the administration reply-to queue window.

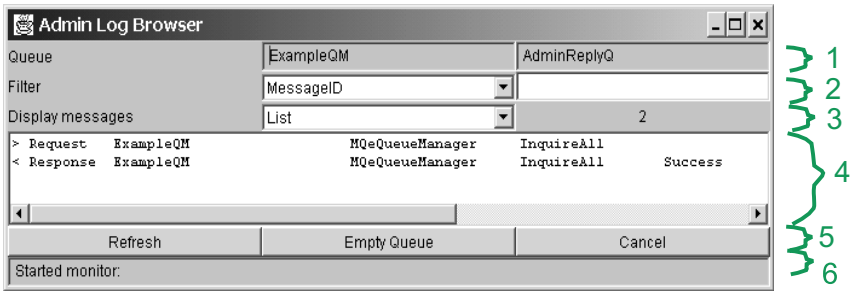

Figure 21. Reply-to queue window

This window has several sections:

1. The name of the administration reply to queue manager and queue

2. Message filter

You can provide a filter to limit the set of messages displayed. This example allows a filter on the *MsgID* and *CorrelID* fields of a message. The example also makes the assumption that the fields contain strings that have been encoded in a byte array.

When administration messages are sent from the example console, the *MsgID* is set to the name of the queue manager to be managed. It is therefore possible to display administration messages only for a specific queue manager.

3. Message view type

You can view messages in the message display panel in the following ways:

List: A one line summary of each message on the queue.

Full: The contents of all messages on the queue.

Both: Two panels, one panel displays a list with a summary line for each message, the other panel displays the contents of a message that has been selected in the message panel.

The number of messages currently being viewed is also displayed.

4. Message display panel

As described in 3, this panel displays messages in various forms. To display a detailed view of a message in a new window, double click the message in the list view.

5. Actions

Several buttons provide actions that are specific to the queue browser:

Refresh

Clears the display and then displays the current contents of the queue. If the queue being browsed is a local queue, a monitor is automatically started. This monitor refreshes the display when new messages are added to the queue. If the queue being browsed is remote then it is not possible to automatically refresh the window when new messages are added. In this case, the **Refresh** button can be used to get the latest contents of the queue.

Empty Queue

Deletes all messages from the queue.

Cancel
Closes the queue browser window.

6. Message
Error and status messages are displayed here.

Action windows

Once you have selected a managed resource type, and you have clicked an action button, a window opens that displays a list of possible parameters for the action. Some parameters are mandatory, others are optional. The following figure shows an example of selecting the add action on a connection:

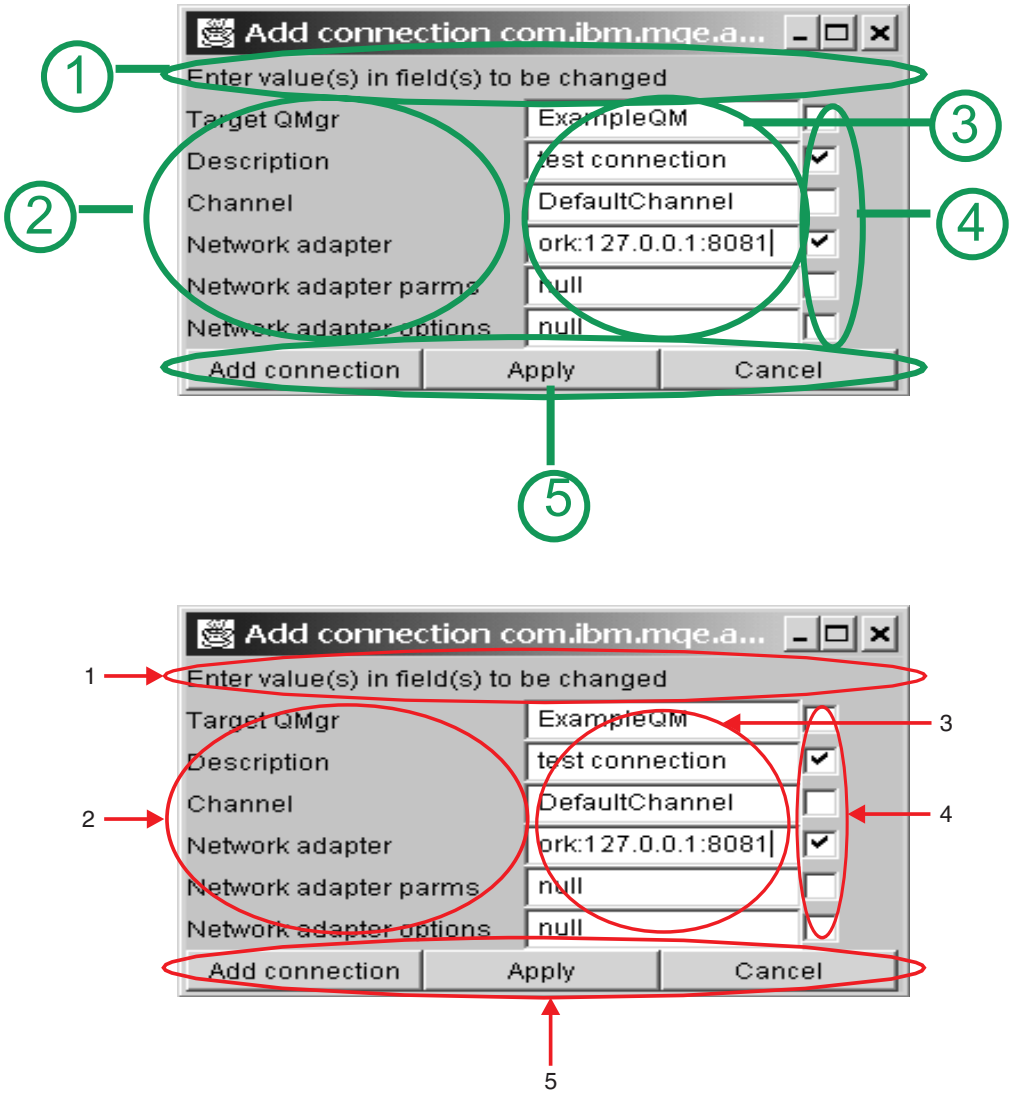

Figure 22. Action window

The action window is the same for most actions. It consists of the following parts:

1. Message area
Error and status messages are displayed here.

2. Names of parameter
Action parameter names.

3. Value of parameter
An input field where you can change the parameter values. The initial value displayed is the default value for the parameter.

4. Send field
The check box for each field is automatically selected when a value is changed. When this field is selected, the field is included in the administration message. By default the administration message only contains values that have changed, it does not contain default values. Default values are understood by the administration message and are not included in the message to ensure that the message size is kept as small as possible. If you change a value back to its default, then you must select the send field check box yourself.

5. Action buttons
For each administration action there are three buttons:

Action The name on this button depends on the administration action (in this example it is **Add connection)**. The action is always to create the administration message and send it to the destination queue manager. The action window is closed.

Apply Create the administration message and send it to the destination queue manager. The action window remains open allowing the same message to be sent multiple times or it can be modified and then sent.

Cancel
Close the action window without sending the administration message.

Reply windows

You can view the outcome of an administration request can be viewed with the administration log browser as described in "Queue browser" on page 114. To see the details of the result of the request, double click on the reply message in the list view.

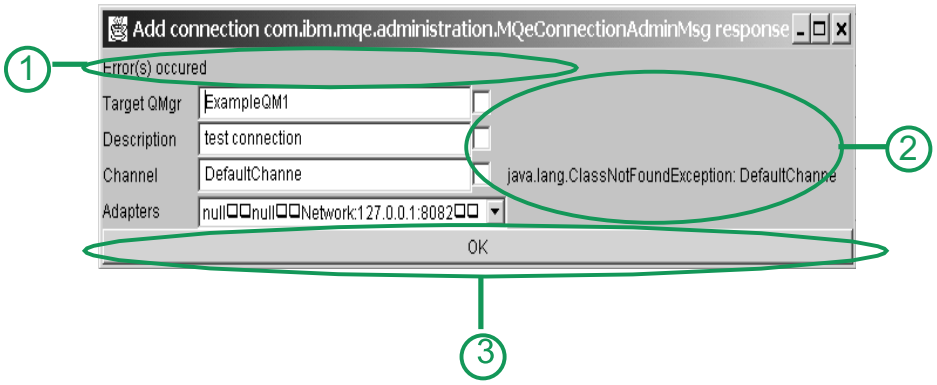

Figure 23. Reply window

The window has the same basic structure as an administration request action window but has the following differences:

1. Message
Displays the return code and result of the action

2. Detailed errors

If the return code was `RC_Mixed`, any errors relating to a particular field are displayed alongside the field.

3. Action buttons

OK Close the action reply window

Chapter 7. MQSeries-bridge

The MQSeries-bridge is a piece of software that allows an MQSeries Everyplace network and an MQSeries network to exchange messages intelligently. Due to the different requirements each aims to satisfy, there are differences in the way the two systems pass messages. The bridge resolves these differences and enables messages to flow between the different systems.

Installation

The bridge code is packaged into the MQeMQBridge.jar file. The class files are also available in the com\ibm\mqe\mqbridge directory. The classpath be set to make the bridge classes accessible when the MQSeries Everyplace server is started. The MQSeries-bridge code runs only on the MQSeries Everyplace gateway platform, not on a device.

MQSeries Java client

The MQSeries-bridge requires that the MQSeries Java Client (version 5.1 or greater) is installed on the MQSeries Everyplace system. The Java client is available as supportpac MA88 for download free from the Web at `http://www.ibm.com/software/ts/mqseries/txppacs/ma88.html`. (The NT client is shipped with MQSeries Version 5.1 for NT.)

Configuring the MQSeries-bridge

The configuration of the gateway requires that some actions be performed on the MQSeries queue manager, and some on the MQSeries Everyplace queue manager. The gateway is logically broken into two pieces, one for each direction of the message (MQSeries Everyplace to MQSeries and MQSeries to MQSeries Everyplace)

The bridge objects are defined in a hierarchy as shown in Figure 24 on page 120

The following rules govern the relationship between the various objects:

- An MQSeries-bridge is associated with a single MQSeries Everyplace queue manager
- A single MQSeries Everyplace queue manager may have more than one bridge object associated with it. You may wish to configure several MQSeries-bridge instances with different routings.
- Each bridge can have a number of MQSeries queue manager definitions
- Each MQSeries queue manager definition can have a number of client connections that allow communication with MQSeries Everyplace
- Each client connection connects to a single MQSeries queue manager, although each service may use a different ″server connection″ on the MQSeries queue manager , or a different set of security, send, and receive exits, ports or other parameters
- A gateway client connection may have a number of ″listeners″ that use that gateway service to connect to the MQSeries queue manager.
- A listener uses only one client connection to establish its connection.
- Each listener connects to a single transmission queue on the MQSeries system.

© Copyright IBM Corp. 2000

- Each listener moves messages from a single MQSeries transmission queue to anywhere on the MQSeries Everyplace network, (using the MQSeries Everyplace queue manager its gateway is associated with). So a gateway can funnel multiple MQSeries message sources through one MQSeries Everyplace queue manager onto the MQSeries Everyplace network.
- When moving MQSeries Everyplace messages to the MQSeries network, the MQSeries Everyplace queue manager creates a number of ″adapter″ objects. Each adapter object can connect to any MQSeries queue manager directly (providing it is configured) and send it's messages to any queue. So a gateway can dispatch MQSeries Everyplace messages routed through a single MQSeries Everyplace queue manager to any MQSeries queue manager.

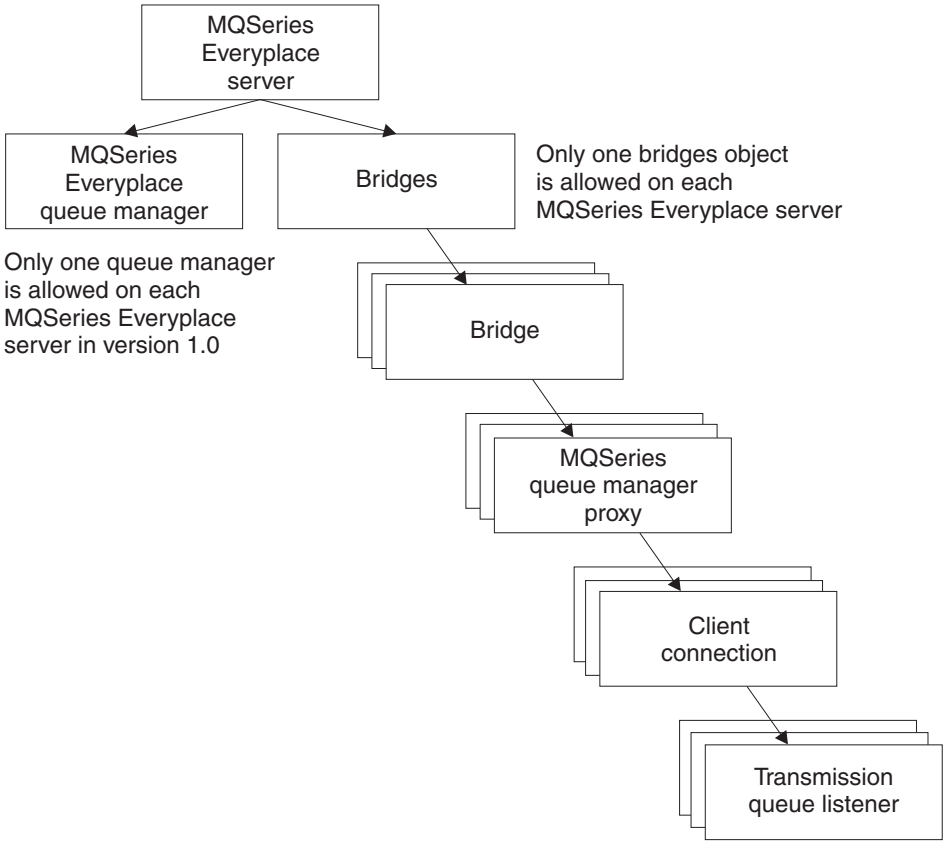

Figure 24. Bridge object hierarchy

Configuring a basic installation

To configure a very basic installation of the MQSeries-bridge you need to complete the following steps:

1. Make sure you have an MQSeries system installed and that you understand local routing conventions, and how to configure the system.
2. Install MQSeries Everyplace on a system (It can be the same system as your MQSeries system is located on if you wish)
3. If the MQSeries Java client is not already installed, download it from the web and install it.(See "MQSeries Java client" on page 119.)
4. Set up your MQSeries Everyplace system and verify that it is working properly before you try to connect it to MQSeries.

5. Update the `MQe_java\Classes\JavaEnv.bat` file so that it points to the java classes that are part of the MQSeries java client, and to the classpath to your JRE (Java Runtime Environment). You need to make sure the "com.ibm.mqbind.jar" and the "com.ibm.mq.jar" classes are in the classpath, and that the 'java\lib' and '\bin' directories are on your path.
6. Plan the routing you intend to implement. You need to decide which MQSeries queue managers are going to talk to which MQSeries Everyplace queue managers.
7. Decide on a naming convention for MQSeries Everyplace objects and MQSeries objects and document it for future use.
8. Modify your MQSeries Everyplace system to define a bridge on your chosen MQSeries Everyplace server. You can use the Administration GUI (examples.mqbridge.awt.AwtMQBridgeServer) to define a bridge.
9. Connect the chosen MQSeries queue manager and the bridge on the MQSeries Everyplace server.

   On the MQSeries queue manager, define one or more java server connection channel so that MQSeries Everyplace can use the java client/bindings to talk to this queue manager. This involves the following steps:

   a. Define the server connection channels
   b. Define a "sync queue" for MQSeries Everyplace use to allow MQSeries Everyplace to provide assured delivery to the MQSeries system. You need one of these per server connection channel that the MQSeries Everyplace system will use.

   On the MQSeries Everyplace server, define an MQSeries queue manager proxy object which holds information about the MQSeries queue manager. Collect the Hostname of the MQSeries queue manager and put it in the MQSeries queue manager proxy object.

   On the MQSeries Everyplace server, define a Client Connection object which holds information about how to use the java client/bindings to connect to the server connection channel on the MQSeries system. Collect the Port number, and all other server connection channel parameters so they match the definition on the MQSeries queue manager.
10. Modify your configuration on both MQSeries Everyplace and MQSeries to allow messages to pass from MQSeries to MQSeries Everyplace.
    a. Decide on the number of routes from MQSeries to your MQSeries Everyplace network. The number of routes you need depends on the amount of message traffic (load) you will be using across each route. If your message load is high you may wish to split your traffic into multiple routes.
    b. Define your routes:
       - Each route requires a transmission queue defined on your MQSeries system. DO NOT define a channel for these transmission queues.
       - Each route requires a matching "transmission queue listener" on your MQSeries Everyplace system.
       - Define a selection of remote queue definitions, (such as remote queue manager aliases and queue aliases) to allow MQSeries message to be routed onto the various MQSeries Everyplace-bound transmission queues you have defined.
11. Modify your configuration on MQSeries Everyplace to allow messages to pass from MQSeries Everyplace to MQSeries:

a. Publish details about all the MQSeries queue managers on your MQSeries network you need to send messages to from the MQSeries Everyplace network. Each MQSeries queue manager requires a "Connections" definition on your MQSeries Everyplace server. All fields except the Queue manager name should be null, to indicate that the normal network is not used to talk to this MQSeries queue manager.

b. Publish details about all the MQSeries queues on your MQSeries network you need to send messages to from the MQSeries Everyplace network. Each MQSeries queue requires an "MQSeries bridge queue" definition on your MQSeries Everyplace server. (This is the MQSeries Everyplace equivalent of a DEFINE QREMOTE).
   - The queue name is the name of the MQSeries queue to which the bridge will send any messages arriving on this MQSeries-bridge queue
   - The queue manager name is the name of the MQSeries queue manager on which the queue is eventually located.
   - The bridge name indicates which bridge should be used to send messages to the MQSeries network
   - The MQSeries queue manager proxy name is the name of the MQSeries queue manager proxy object in the MQSeries Everyplace configuration which can connect to an MQSeries queue manager. The MQSeries queue manager should have a route defined to allow messages to be posted to the "Queue Name" on "Queue Manager Name" to eventually deliver the message to its final destination.

12. Start your MQSeries and MQSeries Everyplace systems to allow messages to flow. The MQSeries system should have its client channel listener started. The MQSeries Everyplace system should have all the objects you have defined started. These can be started explicitly using the Administration GUI, by configuring the rules class to indicate the start-up state (running/stopped) and restarting the MQSeries Everyplace server, or a mixture of the two. The simplest way to manually get objects going is to send the relevant bridge object a "start" command, indicating that all its children and children's children should be started also.
    - To allow messages to pass from MQSeries Everyplace to MQSeries, you need to start the client connection objects in MQSeries Everyplace.
    - To allow messages to pass from MQSeries to MQSeries Everyplace, you need to start both the client connection objects, and the relevant transmission queue listeners.

13. Create transformer classes, and modify your MQSeries Everyplace configuration to use them. A transformer class will convert from a specific MQSeries message format into an MQSeries Everyplace message format, and vice-versa. These format-converters need to be written in java and configured in several places throughout the bridge configuration.

    a. Create java transformer classes
       - Talk to your application developers and find out the message formats of the MQSeries messages which need to pass over the bridge.
       - Write a set of small transformer classes to convert each MQSeries message format into an MQSeries Everyplace message. (See the example transformer for details) OR Write a huge transformer which understands all of the message formats, and can convert between MQSeries and MQSeries Everyplace formats OR Write a transformer class which understands how to recognize an MQSeries message format, and can load and invoke a small transformer to do the conversion of that message. See "Transformers" on page 146.

b. You may wish to replace the default transformer class. Use the administration GUI to ″update″ the Default transformer class parameter in the bridge object's configuration.
c. You may wish to specify a non-default transformer for each MQSeries-bridge queue definition. Use the administration GUI to ″update″ the ″transformer″ field of each MQSeries-bridge queue you have in the configuration
d. You may wish to specify a non-default transformer for each MQSeries transmission queue listener. Use the administration GUI to ″update″ the ″transformer″ field of each listener you have in the configuration
e. Restart the bridge, and listeners.

Sample configuration tool

MQSeries Everyplace systems and the MQSeries-bridge are complex environments and they can be difficult to configure. A sample configuration tool that helps to create an initial configuration is included with the MQSeries-bridge. The source code for the tool is provided and you can subclass it, modify it, and change its behavior as you wish.

This documentation explains what this sample tool does and how to use it.

Limitations

The sample configuration tool cannot be used on a server that has a large number of MQSeries Everyplace queue manager connections defined. For instance, if you have a large number of mobile phones, each with a separate queue manager, and the server had a 'connection' defined for each, then the tool would not work, as it sometimes queries the list of connections. In such situations, the tool stalls and the JVM the wizard runs in fails due to a lack of memory. If you are trying to administer a server that has many connections to other MQSeries Everyplace queue managers, we recommend you use the `examples.mqbridge.administration.console.AdminGateway` application instead.

Steps required to configure the bridge

To configure a very basic installation of the MQSeries-bridge you need to complete the steps in "Configuring a basic installation" on page 120. The sample tool aims to provide a simple way of doing the steps 8-12 in this list.

Configuration example

This section describes an example configuration of 4 systems.

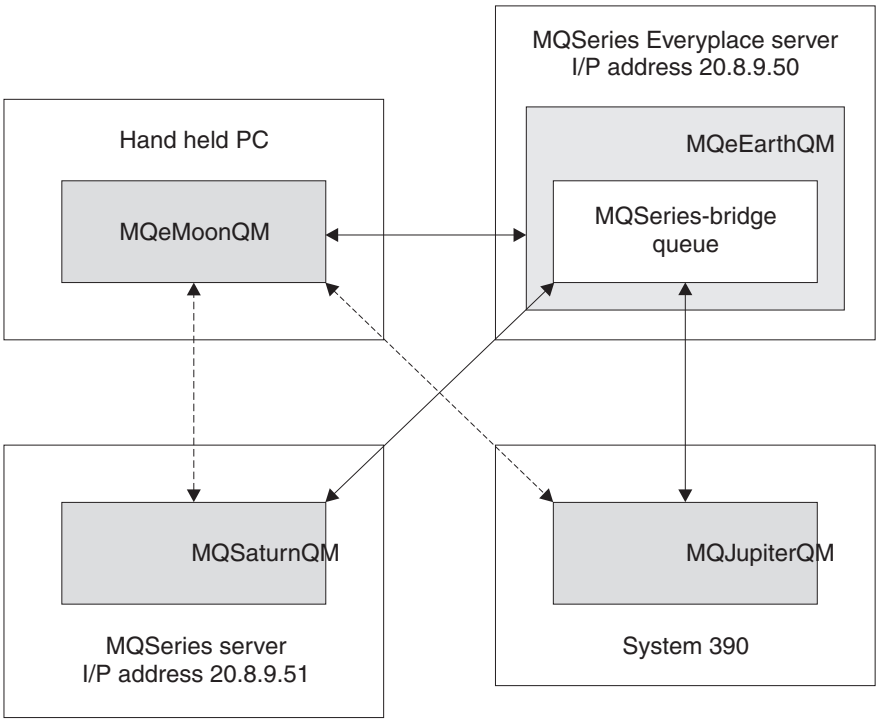

Figure 25. Configuration example

The four systems are:

MQeMoonQM
This is an MQSeries Everyplace client queue manager, sited on a handheld PC. The user periodically attaches the handheld PC to the network, to inter-work with the MQeEarthQM MQSeries Everyplace gateway.

MQeEarthQM
This is on a Windows/2000 machine, with an I/P address of 20.8.9.50 This is an MQSeries Everyplace gateway (server) queue manager.

MQSaturnQM
This is an MQSeries queue manager, installed on a Windows/NT platform. The I/P address is 20.8.9.51

MQJupiterQM
This is an MQSeries queue manager, installed on a System/390 platform.

Requirement

The requirement for this example is that all machines are able to post a message to a queue on any of the other machines.

It is assumed that all machines are permanently connected to the network, except the MQeMoonQM machine, which is only occasionally connected.

Initial setup

For this example, it is assumed that there are local queues, to which messages can be put, on all the queue managers. These queues are called:

- MQeMoonQ on the MQeMoonQM
- MQeEarthQ on the MQeEarthQM
- MQSaturnQ on the MQSaturnQM
- MQJupiterQ on the MQJupiterQM

Enabling MQeMoonQM to put and get messages to and from the MQeEarthQM queue manager

On MQeMoonQM:

Define a connection:

Target queue manager name: MQeEarthQM
Adapter: Network:20.8.9.50

Applications are now be able to use any queue defined on the MQeEarthQM queue manager directly, when the MQeMoonQM is connected to the network. The requirement states that applications on MQeMoonQM should be able to send messages to MQeEarthQ in an asynchronous manner... so we need to define a remote queue definition (rather than rely on auto-descovery) in order to set up the asynchronous linkeage to the MQeEarthQ queue.

Define a remote queue:

Queue name: MQeEarthQ
Queue manager name: MQeEarthQM
Access mode: Asynchronous

Application on MQeMoonQM now have access to the MQeMoonQ (a local queue) in a synchronous manner, and the MQeEarthQ in an asynchronous manner.

Enabling the MQeEarthQM to send messages to the MQeMoonQM queue manager

Since the MQeMoonQM is not attached to the network for most of the time, we will use a store-and-forward queue on the MQeEarthQM to collect messages destined for the MQeMoonQM queue manager.

On MQeEarthQM:

Define a store-and-forward-queue

Queue name: TO.HANDHELDS
Queue Manager Name: MQeEarthQM
(as the queue lives on the MQeEarthQM queue manager)

Add a queue manager to the store-and-forward queue:

Queue Name: TO.HANDHELDS
Queue manager: MQeMoonQM

We also need to set up a (similarly named) home-server queue on the MQeMoonQM queueue manager, which ″pulls″ messages out of the store-and-forward queue and puts them to a queue on the MQeMoonQM queue manager.

On MQeMoonQM:

Define a home-server queue:

Queue Name: TO.HANDHELDS
Queue manager name: MQeEarthQM
(as the home server queue)

Any messages arriving at MQeEarthQM that are destined for MQeMoonQM are stored temporarily in the store-and-forward queue TO.HANDHELDS. When MQeMoonQM next connects to the network, it's home-server queue TO.HANDHELDS gets any messages currently on the store-and-forward queue, and delivers them to the MQeMoonQM queue manager, for storage on local queues.

Applications on MQeEarthQM can now send a message to MQeMoonQ in an asynchronous manner.

Enabling MQeEarthQM to send a message to MQSaturnQ

On MQeEarthQM:

Define a bridge:

Bridge name: MQeEarthQMBridge

Define an MQ queue manager proxy:

Bridge Name: MQeEarthQMBridge
MQ QMgr Proxy Name: MQSaturnQM
Hostname: 20.8.9.51

Define a client connection:

Bridge Name: MQeEarthQMBridge
MQ QMgr Proxy Name: MQSaturnQM
ClientConnectionName: MQeEarth.CHANNEL
SyncQName: MQeEarth.SYNC.QUEUE

Define a connection:

ConnectionName: MQeSaturnQM
Channel: null
Adapter: null

Define an MQBridge queue:

Queue Name: MQSaturnQ
MQ Queue manager name: MQSaturnQM
Bridge name: MQeEarthQMBridge
MQ QMgr Proxy Name: MQSaturnQM
ClientConnectionName: MQeEarth.CHANNEL

On MQSaturnQM:

Define a server connection channel:

Name: MQeEarth.CHANNEL

Define a local "sync" queue:

Name: MQeEarth.SYNC.QUEUE

The sync queue is needed for assured delivery.

Applications on MQeEarthQM can now send a message to the MQSaturnQ on MQSaturnQM.

Enabling MQeEarthQM to send a message to MQJupiterQ

On MQeEarthQM:

Define a connection:

ConnectionName: MQeJupiterQM
Channel: null
Adapter: null

Define an MQBridge queue:

Queue Name: MQJupiterQ
MQ Queue manager name: MQJupiterQM
Bridge name: MQeEarthQMBridge
MQ QMgr Proxy Name: MQSaturnQM
ClientConnectionName: MQeEarth.CHANNEL

On MQSaturnQM:

Define a remote queue definition:

Queue Name: MQJupiterQ
Transmission Queue: MQJupiterQM.XMITQ

On both MQSaturnQM and MQJupiterQM:
Define a channel to move the message from the MQJupiterQM.XMITQ on MQSaturnQM to MQJupiterQM.

Now applictions on MQeEarthQM can send a message to MQJuputerQ on MQJupiterQM, via MQSaturnQM.

Enabling MQeMoonQM to send a message to MQJupiterQ and MQSaturnQ

On MQeMoonQM:

Define a connection:

Target Queue manager name: MQSaturnQM
Adapter: MQeEarthQM

Define a remote queue definition:

Queue name: MQSaturnQ
Queue manager name: MQSaturnQM
Access mode: Asynchronous

The connection indicates that any message bound for the MQSaturnQM queue manager should go through the MQeEarthQM queue manager.

Define a connection:

Target Queue manager name: MQJupiterQM
Adapter: MQeEarthQM

Define a remote queue definition:

Queue name: MQJupiterQ
Queue manager name: MQJupiterQM
Access mode: Asynchronous

Applications connected to MQeMoonQM can now issue messages to MQeMoonQ, MQeEarthQ, MQSaturnQ, and MQJupiterQ, even when the handheld PC is disconnected from the network.

Enabling MQSaturnQM to send messages to the MQeEarthQ

On MQSaturnQM:

Define a local queue:

Queue name: MQeEarth.XMITQ
Queue type: transmission queue

Define a queue manager alias (remote queue definition):

Queue name: MQeEarthQM
Remote queue manager name: MQeEarthQM
Transmission queue: MQeEarth.XMITQ

On MQeEarthQM:

Define a Transmission queue listener:

Bridge name: MQeEarthQMBridge
MQ QMgr Proxy Name: MQSaturnQM
ClientConnectionName: MQeEarth.CHANNEL
Listener Name: MQeEarth.XMITQ

Applications on MQSaturnQM can now send messages using the MQeEarthQM queue manager alias to the MQeEarthQ. This routes each message onto the MQeEarth.XMITQ, where the MQe transmission queue listener MQeEarth.XMITQ gets them, and moves them onto the MQSeries Everyplace network.

Enabling MQSaturnQM to send messages to the MQeMoonQ

On MQSaturnQM:

Define a queue manager alias (remote queue definition):

Queue name: MQeMoonQM
Remote queue manager name: MQeMoonQM
Transmission queue: MQeEarth.XMITQ

Applications on MQSaturnQM can now send messages using the MQeMoonQM queue manager alias to the MQeMoonQ. This routes each message to the MQeEarth.XMITQ, where the MQe transmission queue listener MQeEarth.XMITQ gets them, and posts them onto the MQSeries Everyplace network.

The store-and-forward queue TO.HANDHELDS collects the message, and when the MQeMoonQM next connects to the network, the home-server queue retrieves the message from the store-and-forward queue, and delivers the message to the MQeMoonQ.

Enabling the MQJupiterQM to send messages to the MQeMoonQ

On MQJupiterQM:

Set up remote queue manager aliases for the MQeEarthQM and MQeMoonQM to route messages to MQSaturnQM using normal MQSeries routing techniques.

Now any application connected to any of the queue managers can post a message to any of the MQeMoonQ, MQeEarthQ, MQSaturnQ or MQJupiterQ.

Additional bridge configuration

Trace of the base MQSeries Java Classes is not usually needed, and so is disabled by default. However it is the responsibility of the active trace handler class to initialize MQSeries trace, and an example of how to do this is shipped with the MQSeries Everyplace classes. The example bridge trace class is `examples.mqbridge.awt.AwtBridgeTrace`. This class is automatically instantiated by the bridge administration GUI. Bridge trace messages are supplied in several languages in `examples.mqbridge.trace`..

In addition, MQExceptions are logged to the OutputStreamWriter defined in com.ibm.mq.MQException.log. (System.err by default). Consult the ″MQSeries Using Java″ manual for more information on initializing and configuring base MQSeries trace.

Administration of the MQSeries-bridge

This section contains information on the tasks associated with the administration of the MQSeries-bridge

The example administration GUI application

An example administration GUI is provided with the MQSeries-bridge. It is a subclass of the `examples.administration.console.Admin` example described in “Example administration console” on page 112.

The subclass is called `examples.mqbridge.administration.console.AdminGateway`.

Bridge function cannot execute on a client queue manager, so using this class in conjunction with a client queue manager does not allow the administration of bridge objects on that client queue manager, but it does enable administration of a remote MQSeries-bridge-enabled server queue manager.

To administer a bridge attached to a local queue manager, use the example server program `<java> examples.mqbridge.awt.AwtMQBridgeServer <server_ini_file>` to start an MQSeries Everyplace server.

Figure 26. MQSeries-bridge administration GUI server window

From the server window either of the following options can be used:

- Click the ’Admin’ button to use the `examples.mqbridge.administration.console.AdminGateway` class to administer the local server queue manager, and it’s bridge objects.
- Click on the ″MQBridge setup″ button to invoke the `examples.mqbridge.setup.MQBridgeWizard` example class, as described in “Sample configuration tool” on page 123.

Both examples demonstrate how to programmatically manipulate bridge configuration objects using the bridge-specific administration message sub-classes; MQeMQBridgesAdminMsg, MQeMQBridgeAdminMsg, MQeMQQMgrProxyAdminMsg, MQeClientConnectionAdminMsg, MQeListenerAdminMsg, and MQeMQBridgeQueueAdminMsg.

Bridge administration actions

Run state

Each administered object has a 'run state'. This can be 'running' or 'stopped' indicating whether the administered object is active or not.

When an administered object is 'stopped', it cannot be used, but its configuration parameters can be queries or updated.

If the MQSeries-bridge queue references a bridge administered object that is 'stopped', it is unable to convey an MQSeries Everyplace message onto the MQSeries network until the bridge, MQSeries queue manager proxy, and client connection administered objects are all 'started'.

The 'run state' of the administered objects can be changed using the start/stop actions from the MQeMQBridgeAdminMsg, MQeMQQMgrProxyAdminMsg, MQeClientConnectionAdminMsg or MQeListenerAdminMsg administration message classes.

The actions supported by the bridge administration objects are described in the following sections.

Start action

An administrator can send a 'start' action to any of the administered objects.

The 'affect children' boolean flag affects the results of this action. The 'start' action starts the administered object and all it's children (and children's children) if the "affect children" boolean field is in the message and is set to 'true'. If the flag is not in the message or is set to 'false', only the administered object receiving the start action changes it's run-state. For example, sending 'start' to the bridge and specifying 'effect children' as true causes all proxy, client connection, and listeners that are ancestors, to start. If 'affect children' is not specified, only the bridge is started. An administered object cannot be started unless it's parent administered object has already been started, therefore sending a start event to an administered object causes all the objects higher in the hierarchy to be started if they are not already running.

Stop action

An administered object can be 'stopped' by sending it a stop action. The receiving administered object always makes sure all the objects below it in the hierarchy are stopped before it is stopped itself.

Inquire action

The inquire action queries values from the administered object it is sent to.

If the administered object is in the 'running' state, the values returned on the inquire are those that are currently in-use. The values returned from a 'stopped' object reflect any recent changes to values made by an 'update' action. Thus, a sequence of start, update, inquire returns the values configured *before* the update, while start, update, stop, inquire returns the values configured *after* the update.

You may find it less confusing if you stop any administered object before updating variable values.

Update action

The update action updates one or more values for characteristics for an administered object. The values set by an 'update' action do not become current until the administered object is next stopped. (See "Inquire action" on page 130.)

Delete action

The delete action permanently removes all current and persistent information about the administered object. The 'affect children' boolean flag affects the outcome of this action. If the 'affect children' flag is present and set to 'true' the administered object receiving this action issues a 'stop' action, and then a 'delete' action to all the objects below it in the hierarchy, removing a whole piece of the hierarchy with one action. If the flag is not present, or it is set to false, then the administered object deletes only itself, but this action cannot take place unless all the objects in the hierarchy below the current one have already been deleted.

Create action

The create action creates an administered object. The 'run state' of the administered object created is initially set to 'stopped'.

Shutting down an MQSeries queue manager

We recommend that before you stop the MQSeries queue manager, you issue a `STOP` administration message to all the MQSeries queue-manager-proxy administered bridge objects, to stop the MQSeries Everyplace network using the MQSeries queue manager. Stopping the MQSeries queue-manager-proxy bridge object prevents any MQSeries Everyplace activity from interfering with the shutdown of the MQSeries queue manager. (This can also be done by issuing a single `STOP` admin message to the MQebridges object.)

If you do not stop the MQSeries queue-manager-proxy bridge object before you shut the MQSeries queue manager, the behavior of the MQSeries shutdown and the MQSeries bridge depends on the type of MQSeries queue manager shutdown you choose, immediate shutdown or controlled shutdown.

Immediate shutdown

Stopping an MQSeries queue manager using immediate shutdown severs (by force) any connections that the bridge has to the MQSeries queue manager (this applies to connections formed using either the Java bindings, or Java client channels). The MQSeries the system shuts down as normal.

This causes all the bridge transmission queue listeners to stop immediately, each one warning that it has shut down due to the MQSeries queue manager stopping immediately.

Any MQSeries-bridge queues that are active retain a (broken) connection to the MQSeries queue manager until:

- The connection times-out, after being idle for an idle timeout period (as specified on the client-connection bridge object), at which point the broken connection is closed.
- The MQSeries-bridge queue is told to perform some action, such as put a message to MQSeries, that attempts to use the broken connection. The `putMessage` operation fails and the broken connection is closed.

MQSeries queue manager shutdown

When an MQSeries-bridge queue has no connection, the next operation on that queue causes a new connection to be obtained. If the MQSeries queue manager is not available, the operation on the queue fails synchronously. If the MQSeries queue manager has been re-started after the shutdown, and a queue operation, such as `putMessage`, acts on the bridge queue, then a new connection to the active MQSeries queue manager is established, and the operation executes as expected.

Controlled shutdown

Stopping an MQSeries queue manager using the controlled shutdown does not sever any connections by force, it waits until all connections are closed (this applies to connections formed using the Java bindings, or Java client channels). Any active bridge transmission queue listeners notice that the MQSeries system is quiescing, and stop with a relevent warning.

Any MQSeries-bridge queues that are active retain a connection to the MQSeries queue manager until:

- The connection times-out, after being idle for an idle timeout period (as specified on the client connection bridge object), at which point the broken connection is closed, and the controlled shutdown of the MQSeries queue manager completes.
- The MQSeries-bridge queue is told to perform some action, such as put a message to MQSeries, that attempts to use the (broken) connection. The `putMessage` operation fails, the broken connection is closed, and the controlled shutdown of the MQSeries queue manager completes.

The bridge client-connection object maintains a pool of connections, that are awaiting use. If there is no bridge activity, the pool retains MQSeries client channel connections until the connection idle time exceeds the idle timeout period (as specified on the client connection object configuration), at which point the channels in the pool are closed.

When the last client channel connection to the MQSeries queue manager is closed, the MQSeries controlled shutdown completes.

Administered objects and their characteristics

This section describes the characteristics of the different types of administered objects associated with the MQSeries Everyplace to MQSeries-bridge. Characteristics are object attributes that can be queried using an `inquireAll()` administration message. The results can be read and used by the application, or they can be sent in an update or create administration messages to set the values of the characteristics. Some characteristics can also be set using the create and update administration messages. Each characteristic has a unique label associated with it and this label is used to set and get the characteristic value.

The following lists show the attributes that apply to each administered object. The attributes are described in detail in alphabetical order in “Attribute details” on page 134. The label constants are defined in com.ibm.mqe.mqbridge.MQeCharacteristicLabels

Characteristics of bridges objects

- Run-state
- Children
- Child

Characteristics of bridge objects

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- HeartBeatInterval
- DefaultTransformer

Characteristics of MQSeries queue manager proxy objects

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- HostName

Characteristics of client connection objects

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- ClientConnectionName
- Port
- AdapterClass
- MQUserID
- MQPassword
- SendExit
- ReceiveExit
- SecurityExit
- CCSID
- SyncQName
- SyncQPurgerRulesClass
- MaxConnectionIdleTime
- SyncQPurgeInterval

Characteristics of MQSeries transmission queue listener objects

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass

- BridgeName
- MQQMgrProxyName
- ClientConnectionName
- ListenerName
- DeadLetterQName
- ListenerStateStoreAdapter
- UndeliveredMessageRuleClass
- TransformerClass

Attribute details

Attribute:
AdapterClass

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_ADAPTER_CLASS`

Valid actions
Inquire, create, update

Description
This is either a java class name, or an alias that can be resolved into a java class name. It is used by the gateway slave.

If not specified, a default of com.ibm.mqe.mqbridge.MQeMQAdapter is used. This parameter is not validated.

Attribute:
AdministeredObjectClass

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_ADMINISTERED_OBJECT_CLASS`

Valid actions
Inquire, create, update

Description
The name of the bridge.

Valid characters are: '0–9' 'A-Z' 'a-z' - . % /

Attribute:
BridgeName

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_BRIDGE_NAME`

Valid actions
Inquire, create, update, delete, start, stop

Description
If you use a symbolic name, it may take longer to detect that this machine is not switched on, or that the name server is not working. If this causes a problem, you can use the actual I/P address in this field instead.

Note: This characteristic is settable only once, when the create administration message is used. Thereafter it is used to identify which bridge administered object should be inquired on, updated, deleted, started, or stopped.

Attribute:
CCSID

Type: Int

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CCSID`

Valid actions
Inquire, create, update

Description
See the MQSeries Using Java documentation for a description of this parameter.

Valid values are: 0..MAXINT, default is 0.

Attribute:
Child

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CHILD`

Valid actions
Inquire

Description
A field containing the name of an MQSeries-bridge administered object.

Attribute:
Children

Type: MQeFields array

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CHILDREN`

Valid actions
Inquire

Description
An array of Child fields, each element containing a Child attribute.

Attribute:
ClientConnectionName

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CLIENT_CONNECTION_NAME`

Valid actions
Inquire, create, update, delete, start, stop

Description

Note: This characteristic is settable only once, when the create administration message is used. Thereafter it is used to identify which bridge administered object should be inquired on, updated, deleted, started, or stopped.

Attribute:
DeadLetterQName

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_DEAD_LETTER_Q_NAME`

Valid actions
Inquire, create, update

Description
If the gateway finds it cannot deliver a message from MQSeries to MQSeries Everyplace (for example, due to a size restriction on the target MQSeries Everyplace queue) then the message cannot be processed by the gateway, and it is placed on a dead letter queue on the MQSeries system. This parameter defines which queue the erroneous message is delivered to.

If not specified, the value of ″SYSTEM.DEAD.LETTER.QUEUE″ is used.

Attribute:
DefaultTransformer

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_DEFAULT_TRANSFORMER`

Valid actions
Inquire, create, update

Description
The classname specified here is used as the ″default transformer class″. When a message is sent from MQSeries to MQSeries Everyplace, the target queue may have a transformer class defined. If it is NOT defined, then this class is used to transform the MQSeries message into MQSeries Everyplace format.

When a message is sent from MQSeries Everyplace to MQSeries, again, the transmission queue listener moving the message onto MQSeries Everyplace may have a transformer class defined. If it is NOT defined, then this class is used to transform the MQSeries message into the MQSeries Everyplace format.

No validation of the value in this field is performed.

Default value is `com.ibm.mqebridge.MQeBaseTransformer`

Attribute:
HeartBeatInterval

Type: Int

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_HEARTBEAT_INTERVAL`

Valid actions
Inquire, create, update

Description

A time interval, expressed in units of 1 minute. 1<=value<=60. The bridge uses a ″heartbeat″ internally to provide a regular stimulus to other administered objects. The administered objects perform small tasks when the heartbeat event arrives, such as 'The client connection will reap old or stale MQSeries connections' or 'the sync queue will be purged'. The heartbeat provides a ″granularity″ of timers which is indivisible, that is, the lower this value is set, the more accurate any actions which compare against the current time will be. For instance, if you say ″reap all MQSeries connections if they have been idle for more than 10 minutes″, but set the heartbeat interval for 3 minute intervals, then an idle MQSeries connection will be checked after 3,6,9 and 12 minutes, but will only be ″reaped″ on the 12th minute. Setting this value lower increases the accuracy of the timer-related heartbeat events, but does so at the cost of efficiency. The more heartbeat events created, the more work is done.

The default value is 5 minutes.

Attribute:

Hostname

Type: Unicode

Label:

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_HOST_NAME`

Valid actions

Inquire, create, update

Description

Used to create connections to this MQSeries queue manager using the MQSeries Java classes. If not specified, then the MQSeries queue manager is assumed to be on the same machine as the JVM, so the java bindings are used to communicate with the MQSeries system.

Note: A blank value is NOT the same as specifying ″localhost″. If a blank value is used, then the bridge uses the MQSeries java bindings which communicate with MQSeries directly (which is faster). If you specify ″localhost″, although it has exactly the same effect, it causes the bridge to use the MQSeries java client classes, which means all communication with MQSeries is through the network (TCP/IP) stack.

The value specified here is not validated at all. If you use a symbolic name, it may take longer to detect that this machine is not switched on, or if the name server is not working. You can instead use the I/P address notation in this field if this causes problems.

Attribute:

ListenerName

Type: Unicode

Label:

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_LISTENER_NAME`

Valid actions

Inquire, create, update, delete, start, stop

Description

The name of this listener. The listener name is the name of the transmission queue on MQSeries that the listener takes messages from. The

combination of MQ_queue_manager_name and MQ_transmission_queue_name pair must be unique across all the gateways that exist.

Note: This characteristic is settable only once, when the create administration message is used. Thereafter it is used to identify which bridge administered object should be inquired on, updated, deleted, started, or stopped.

Attribute:
ListenerStateStoreAdapter

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_LISTENER_STATE_STORE_ADAPTER`

Valid actions
Inquire, create, update

Description
In order to provide assured message delivery of persistent messages, the listener class uses an adapter to store state information. This is the class name (or alias of the classname) of the adapter that is loaded to manage the storing and recovery of the state information to and from disk. Two adapters are currently supported- com.ibm.mqe.adapters.MQeDiskFieldsAdapter (which stores state information on the local filesystem) and com.ibm.mqe.mqbridge.MQeMQAdapter (which stores state information on the MQSeries server). The disk adapter is generally quicker than using the MQSeries-based adapter. The classname can be followed by a colon separated list of arguments, although only the MQeDiskFieldsAdapter uses them. In this case MQeDiskFieldsAdapter can be followed by a colon and a fully qualified path name to a file that contains the state information. For example, in order to use the disk fields adapter to store the listener's state information in the file `c:\folder\state.sta`, the listener-state-store-adapter field should contain the value "com.ibm.mqe.Adapters.MQeDiskFieldsAdapter:c:\folder\state.sta". A file specified by this parameter need not exist already. If the supplied path name ends in a folder separator (for example. '\' in DOS) then it is assumed that the supplied parameter is a directory, and a state file called '<ListenerName>-listener.sta' is created inside it (where <ListenerName> is the name of the listener, from the registry entry). If no path name is supplied, the listener uses a file called '<ListenerName>-listener.sta' inside the current Java working directory If the MQeMQAdapter is being used, no additional arguments are required.

The default value of the ListenerStateStoreAdapter field is "com.ibm.mqe.Adapters.MQeDiskFieldsAdapter".

Attribute:
MaxConnectionIdleTime

Type: Int

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_MAX_CONNECTION_IDLE_TIME`

Valid actions
Inquire, create, update

Description

Each client connection object in the bridge maintains a ″pool″ of MQSeries Java client connections to its MQSeries system.

When an MQSeries connection becomes idle, through lack of use, whether it is in the pool or not, a timer is started. If the timer reaches the current value of this parameter, then the idle connection is thrown away. This is known as *reaping* the connection. This is done inorder to save resources when the connection is idle. The connection pool is an efficiency device used within the MQSeries bridge. Creation of new MQSeries client connections is a resource intensive operation. If there are idle connections in the pool, one of these is reused, thus saving time and resources by avoiding a ″get new connection″ operation. The higher the MaxConnectionIdleTime value, the more likely there is going to be an idle connection waiting in the connection pool, but the more client connections there are idle, and consuming resources in the JVM. Setting this value lower, decreases the likelihood of an idle connection being available, but also decreases the number of idle connections , so less resources are consumed.

The time is expressed in units of 1 minute.

The Valid range: 0 <= value <= 720 (12 hours) and the default is 5 (minutes).

Setting this value to 0 effectively means ″don't use a connection pool″ and whenever an MQSeries client connection is idle, it is reaped or discarded. This is an inefficient way of using this parameter.

The granularity of any timeouts specified in the bridge are only checked with a timer granularity equal to the `heartbeatInterval` bridge parameter.

MaxConnectionIdleTime can have a direct effect on the length of time it takes to shut down an MQSeries Everyplace system. See for more details "Shutting down an MQSeries queue manager" on page 131.

Attribute:

MQPassword

Type: Unicode

Label:

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_PASSWORD`

Valid actions

Inquire, create, update

Description

Used by the java client. The password field on the MQSeries calls is set to ″″ if this attribute is not specified. The value you specify here overrides any defaults used. This parameter is not validated.

Attribute:

MQQMgrProxyName

Type: Int

Label:

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_MQ_Q_MGR_PROXY_NAME`

Valid actions

Inquire, create, update, delete, start, stop

Description

Note: This characteristic is settable only once, when the create administration message is used. Thereafter it is used to identify which bridge administered object should be inquired on, updated, deleted, started, or stopped.

Attribute:
MQUserID

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_USER_ID`

Valid actions
Inquire, create, update

Description
Used by the java client. The user-id field on the MQSeries calls is set to "" if this is not specified. The value you specify here overrides any defaults used. This parameter is not validated.

Attribute:
Port

Type: Int

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_PORT`

Valid actions
Inquire, create, update

Description
Used to create connections to this MQSeries queue manager using the MQSeries Java classes. If not specified, then the MQSeries queue manager is assumed to be on the same machine as the JVM, so the java bindings are used to talk to the MQSeries system.

Valid range 0..MAXINT.

Attribute:
ReceiveExit

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_RECEIVE_EXIT`

Valid actions
Inquire, create, update

Description
Used to match the exit used at the other end of the Client channel.

This parameter is not validated.

Attribute:
Run-state

Type: Int

Label:

com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_RUN_STATE

Valid actions
Inquire

Description
Indicates whether the administered object is 'running' (value=1) and hence in-use, or 'stopped' (value=0) and hence not in use. When an object is stopped it can have its properties changed dynamically.

Attribute:
SecurityExit

Type: Unicode

Label:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SECURITY_EXIT

Valid actions
Inquire, create, update

Description
Used to match the exit used at the other end of the Client channel.

This parameter is not validated.

Attribute:
SendExit

Type: Unicode

Label:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SEND_EXIT

Valid actions
Inquire, create, update

Description
Used to match the exit used at the other end of the Client channel.

This parameter is not validated.

Attribute:
StartupRuleClass

Type: Unicode

Label:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_STARTUP_RULE_CLASS

Valid actions
Inquire, create, update

Description
* This is a rule class that is used when the administered object is loaded at system start-up, or when it is first created. The rule class name is not validated. The rule class specified dictates whether the administered object is started, and whether or not its children are started. The default rule is com.ibm.mqe.mqbridge .MQeStartupRule This default causes the administered object to start, and all its parents to start up. Setting this field to "" (blank) causes the administered object not to be started. See "MQeStartupRule" on page 152

Attribute:
SyncQName

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SYNC_Q_NAME`

Valid actions
Inquire, create, update

Description
The name of the sync queue that is used by the MQSeries-bridge on this MQSeries queue manager. Valid characters forming the name are: '0'-'9' 'A'-'Z' 'a'-'z' '_' '.' '%' '/' The sync queue is an MQSeries queue that is used to keep track of which messages are in the process of moving from MQSeries Everyplace to MQSeries. If a message is part way through the logic that assures the once-only delivery of a message, then there will be another message on the sync queue, indicating how far through the logic the message has progressed. If the MQSeries Everyplace system is shut down cleanly, then the sync queue should be empty. If the MQSeries Everyplace system crashes, then some persistent state information is left in the sync queue. This information is used when the MQSeries Everyplace system restarts so that it continues from where it left off. The name of the sync queue can be the same for client connections on the same bridge, or on different bridges, providing the send, receive and security exits used when talking to that sync queue are the same. The sync queues must exist on the MQSeries queue manager for MQSeries Everyplace->MQSeries message transfer to work. If the listener state class is the MQeMQAdapter, it means that this sync queue is used for storing persistent state information about the listeners also, so it must exist on MQSeries for the listeners, in order to move MQSeries messages to MQSeries Everyplace. The listener does not use this parameter if the state information is being stored by an MQeDiskFieldsAdapter. We recommended a naming scheme of: MQE.SYNCQ.<ClientConnectionName> so that you know which client connection is using which sync queue.

The default is "MQE.SYNCQ.DEFAULT".

Attribute:
SyncQPurgeInterval

Type: int

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SYNC_Q_PURGE_INTERVAL`

Valid actions
Inquire, create, update

Description

Attribute:
SyncQPurgerRulesClass

Type: Unicode

Label:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SYNC_Q_PURGER_RULES_CLASS`

Valid actions
Inquire, create, update

Description

The name of the rules class used when a message on the sync queue indicates a failure of MQSeries Everyplace to confirm a message.

The default is a classname that just reports the condition in the MQSeries Everyplace trace.

This parameter is not validated.

Attribute:

TransformerClass

Type: Unicode

Label:

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_TRANSFORMER`

Valid actions

Inquire, ceeate, update

Description

This is the name of the java class that is used to convert the MQSeries message into an MQSeries Everyplace message. When a message is taken from MQSeries by the listener, it is transformed into an MQSeries Everyplace format message using the specified transformer. If the transformer class is specified as ″null″ or a blank string, then the DefaultTransformerClass parameter provided on the bridge configuration parameters is used as the transformer. If the default is also set to null/blank, then messages cannot be transferred.

The default value is ″″

See “Transformers” on page 146 for more details.

Attribute:

UndelivedMessageRuleClass

Type: Unicode

Label:

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_UNDELIVERED_MESSAGE_RULE_CLASS`

Valid actions

Inquire, create, update

Description

The name of the MQeUndeliveredMessageRule class. When a message moving from MQSeries to MQSeries Everyplace cannot be delivered, this rule class is consulted to decide what action the listener should take. The rule tells the listener to wait and retry, shut down or deal with the message as defined in the MQMessage report options.

The default value is: `com.ibm.mqe.mqbridge.MQeUndeliveredMessageRule`.
See “MQeUndeliveredMessageRule” on page 151.

How to send a test message from MQSeries to MQSeries Everyplace

There are many ways of arranging your routing on the MQSeries system to test the transmission of a message. One method is to use the bridge setup wizard tool to define queue manager aliases for each MQSeries Everyplace queue manager that it knows about. This document describes how to use the resultant configuration to send a message to the MQSeries Everyplace queue.

sending a bridge test message

1. Select the MQSeries First Steps program from the MQSeries Client v 5.1
2. Select the API exerciser from the First Steps screen
3. In the API Exerciser Queue Managers screen:
   - Select the MQSeries queue manager to which the bridge is connected. (The example is called MQA)
   - Select the 'Advanced mode' checkbox
   - Select MQCONN button
   - Select the 'Queues' tab to display the Queues screen
   - Select MQOPEN to display the 'MQOPEN Selectable Options' screen
4. In the MQOPEN Selectable Options screen:
   - Make sure the MQOO_INPUT_AS_Q_DEF is NOT selected
   - Make sure the MQO_OUTPUT is selected
   - Fill in the ObjectName field with the name of the queue, on the MQSeries Everyplace queue manager that you wish the message to go to. (The example is called Q1)
   - Fill in the ObjectQMgrName field with the name of the MQSeries Everyplace queue manager you wish the message to go to. (The example is called ExampleQM)
   - Click on OK to open a route to the queue.
5. In the API Exerciser Queues screen:
   - Select the MQPUT button to display the 'MQPUT -Argument Options' screen
6. In the MQPUT - Argument Options screen:
   - Type in your message
   - Click on OK to send the message to Q1 on ExampleQM on the MQSeries Everyplace system

Dead-letter queues

MQSeries Everyplace has a similar concept of dead-letter queues to MQSeries. Such queues store message that cannot be delivered. However, there are important differences in the manner they are used.

In MQSeries, if a message is being moved from queue manager A to queue manager B, then if the channel connecting A to B cannot deliver the message, the message can be placed on the *receiving queue manager's* (B's) dead-letter queue.

Due to the nature of the MQSeries Everyplace architecture, if a message is being sent from queue manager A to queue manager B, but it cannot be delivered, the message can be placed on the *sending queue manager's* (A's) dead letter queue. The behaviour is controlled by customizeable rules, see "MQeUndeliveredMessageRule" on page 151 for more details.

The MQSeries-bridge's transmission queue listener is analogous to an MQSeries channel, pulling messages from an MQSeries transmission queue, and delivering them to the MQSeries Everyplace network. It follows the MQSeries Everyplace convention in that if a message cannot be delivered, an *undelivered message rule* is consulted to determine how the transmission queue listener should react. If the rule indicates the report options in the message header, and these indicate that the message should be put onto a dead-letter queue, the message is placed on the MQSeries queue (on the sending queue manager).

putMessage() considerations for the MQSeries-bridge

If an application uses `putMessage`, specifying that a `confirmputMessage()` should not be used to confirm this message (confirm parameter==false), the MQSeries-bridge does not used assured delivery logic to pass the message to MQSeries. It does a simple ″MQPut″ to the MQSeries queue to which the message is being routed. If the MQSeries system, bridge, or any of the MQSeries Everyplace systems between the caller of the message, and the MQSeries system fail, then the application is unable to determine whether the message has been sent or not. The application may wish to then resend the message, possibly resulting in two identical messages arriving on the MQSeries queue.

If this causes a problem, the application programmer should choose to use the `putMessage()` and `confirmputMessage()` calls instead.Using the `putMessage()` with the confirm parameter=true causes the bridge to use assured delivery logic to put the message to the MQSeries system.

If any component of the path between the MQSeries system and the sending application fails, the application programmer is unable to determine whether the message got to it's destination or not. In this case, the application should take the original message, and add a boolean MQeField to it. For example:

```
msg.putBoolean( MQe.Qos_Retry)
```

To indicate that this message has been sent in the past. The message is can then be issued (again) using the `putMessage()` method. The MQSeries-bridge uses it's assured delivery logic to assure that only one of the two `putMessage()` calls actually put a message to MQSeries.

If the second `putMessage()` is issued without the MQe.Qos_Retry, then either one or two messages containing the same data may end up in the target queue. Thus, you must always use the Qos_Retry boolean field if you may have ever attempted to send the message in the past.

If the `putMessage()` is used, without the confirm flag being set, and a successful return code is received, the application can be assured that the message has been passed to the MQSeries queue.

If the `putMessage()` is used, with the confirm flag being set, and a successful return code is received, the application can be assured that the message has been passed to the MQSeries queue, but the bridge retains some information about the message (on it's ″sync queue″) so that it is able to prevent duplicate messages being sent by the application. The bridge only prevents duplicate messages being sent if they have the Qos_Retry bit set. The `confirmputMessage()` flushes this ″memory″ of the message from the bridge (from the ″sync queue″).

The following procedure causes four messages to arrive on the MQSeries queue.

	create a new message	
(1)	`putMessage(Confirm=Yes)`	- Causes the message to be delivered to MQ, but some note made on the sync Q
	set the retry bit on the message	
	`putMessage(Confirm=Yes)`	- Supressed, as the message is already noted in the sync Q
	`putMessage(Conirm=Yes)`	- Supressed, as the message is already noted in the sync Q

bridge - putMessage considerations

(2)	`putMessage(Confirm=No)`	- NOT supressed, only puts using confirm=yes will be supressed using the sync Q. Msg delivered to the MQSeries Queue.
	remove the retry bit from the message	
(3)	`putMessage(Confirm=Yes)`	- Causes the message to be sent to MQ, the retry bit was not set, so the bridge did not look at it's sync Q
	`ConfirmputMessage()`	- Causes the bridge to clear it's memory of the message
	set the retry bit on the message	
(4)	`putMessage()`	- Causes the message to be sent

Transformers

A transformer is a Java class that is capable of converting an MQSeries Everyplace message into an MQSeries message, and is capable of converting an MQSeries message into an MQSeries Everyplace message. Transformers are derived from the MQeBaseTransformer class.

The transformer can be specified in several ways during the MQSeries-bridge configuration.

- A ″Default transformer″ can be specified for each MQSeries-bridge
- A transformer can be specified for each MQSeries-bridge queue
- A transformer can be specified for each MQSeries transmission queue listener

In each case the MQSeries-bridge configuration expects a java class name, or an MQSeries Everyplace alias that resolves into a java class. The resulting java class must be derived from the MQeBaseTransformer class.

The transformer is responsible for all aspects of message conversion so it must be written by the end-user programmer to provide a method of converting between the MQSeries-native message format, and the MQSeries Everyplace-native message format. This means that whenever you create a new format for messages that flow between MQSeries and MQSeries Everyplace, you need to create or modify a transformer class for the new message format.

These changes can be handled in a variety of ways:

- Write a huge transformer which can convert all your message formats.

  This could be implemented using the inheritance model of Java, where one transformer inherits from another, which inherits from another thus forming a chain of transformers, or it could be implemented as one huge java class.

  Positive aspects of this approach are:

  – You can change the ″default transformer″ specified for an MQSeries-bridge. This requires only one point of configuration to determine the transformer to use for all operations. (Leave the MQSeries-transmission-queue-listener and the MQSeries-bridge queue definitions blank/null where the transformer name is specified)

  – This a very simple approach

  Negative aspects of this approach are:

  - When formats of an application change, or when a new format is invented, this large transformer has to be changes and redeployed everywhere.
  - It may not be possible to create one transformer that understands all the message formats in your system

- Write a series of medium-sized transformers, each being capable of understanding and transforming various groups of message formats.

  Each transformer may be responsible for working with a specific application and the MQSeries Everyplace routing may be set up such that each application has exclusive use of a set of MQSeries bridge queues, and MQSeries-transmission-queue-listeners. The transformer name on the MQSeries-bridge queues and transmission-queue-listeners are then set to be application-specific.

  Positive aspects of this approach are:

  - The programmer has complete control of where messages are routed, and can make sure that the correct transformer is used.
  - The approach is simple
  - If you add or change a message format the transformer only need to be changed along the path that the changed or new message formats can flow

- Write a separate transformer for every message format in your system.

  This requires that a higher-level transformer is created (see examples.mqbridge.transformers.MQeListTransformer)that uses a list of these very small transformers, invoking each in turn until the a transformer that can use the message is found.

  Each transformer has knowledge of a single message format.

  Care must be taken with each message format, and transformer to make sure each small transformer is able to uniquely identify the format of the messages that it transforms. Do not allow an instance of a message to be transformable by more than one transformer. Each transformer must be able to examine each message to determine whether the message is in the format that the transformer was designed to work on.

  Various list transformers may be used at different points in the MQSeries-bridge configuration. At the most basic level, creating a list transformer with a huge list of all the small transformers available, and setting this to be the default. At the most complex level, creating a list transformer with a very small list of transformers, and setting MQSeries-bridge queue and MQSeries-transmission-queue-listener transformer parameters.

  The list transformer may obtain its list from hard-coded literal string constants within the java source code itself, from the system environment variables of the JVM, from the underlying operating system environment, from an ASCII data file that is loaded when the list transformer class is loaded, or simply by looking at which transformer classes are available in the file system when it is loaded. The choice of methods is left to the end-user programmer. The example list transformer uses the method of hard-coding the transformer list in its java source code.

  Positive aspects of this approach

  - This approach is more object oriented, allowing the knowledge about a particular message format to be completely encapsulated within a single small transformer, while the ″list″ transformer only understands which transformers are available.
  - Adding a new small transformer need not cause a list transformer to change. For example, if the list transformer looks at the file system to see which

transformers are available, then simply adding the transformer to the correct location in the file system may be enough to cause the transformer to be used.

- Use a mixture of all of the above methods.

The examples.mqbridge.transformers.MQeListTransformer example transformer class

This example transformer demonstrates how a higher-level transformer class can use a list of very small transformers in order to perform message transformation, without itself having any knowledge of the format of the message.

The source file is `examples\mqbridge\transformers\MQeListTransformer.java` and is a simple MQSeries to MQSeries Everyplace transformer class.

This transformer does not actually understand the format of any messages that is passed to it. It has an ordered list of smaller transformers. When a message needs to be transformed, this class works through its list of transformers one by one, presenting the message to each transformer. The results of the first transformer to successfully return a converted message are returned to the user of this class.

This style of transformer could be used in conjunctions with a collection of smaller transformers, where each ″small″ transformer understands a limited number of message formats.

This class keeps its list of transformers in a static ordered list (array) but it could easily have read the list from a file when the activate method is called, or obtained a list through some other method (possibly using the user-defined parameters passed on the activate method to do so).

To use the example, write a series of small transformers, and put their class names into the static list at the top of the example file. Compile, and set the resulting (high-level) transformer into the required places in the bridge configuration.

Transformers and expiry time considerations

Special care needs to be taken when converting the expiry times between MQSeries and MQSeries Everyplace.

MQSeries Everyplace expiry times are specified as either an explicit time after which the message expires, or a delta in units of 1 millisecond of how long after the message creation time the message will expire.

MQSeries units are in 1/10ths of a second.

Failure to convert these expiry times in your transformer can result in messages expiring, and apparently being ″lost″.

MQSeries style messages

MQeMQMsgObject is a subclass of MQeMsgObject that supports MQSeries style messages within MQSeries Everyplace. It is typically used to exchange messages with MQSeries applications using the default transformer in the MQSeries-bridge. This generates an MQeMQMsgObject when it receives a standard MQSeries message. Similarly, if an MQSeries Everyplace application generates an MQeMQMsgObject and sends it to MQSeries, the default transformer in the bridge knows how to transform it into a standard MQSeries message.

If the MQeMQMsgObject class does not meet your requirements, you can write a transformer for the bridge that uses another type of message object more suited to your application.

To save space, this class can be removed from systems that do not intend to use it.

Reading an MQSeries style message

When an application receives a message, it can check whether the message belongs to the MQeMQMsgObject class as follows:

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
  MQeMsgObject msg = MyQM.getMessage(qmgr, queue, null, null, 0);
  if (msg instanceof MQeMQMsgObject)
  {
    MQeMQMsgObject mqeMsg = (MQeMQMsgObject) msg;
    ...
  }
```

If the message does belong to this class, all the information from the MQSeries message header can be accessed as well as the message data by using the appropriate get methods on the message object. The header information can be obtained using methods of the form `getxxx()` where *xxx* is the name of the header field. For consistency, the names and types of the header fields follow those of the MQSeries Java Client. The application data is obtained using the `getData()` method.

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
  if (msg instanceof MQeMQMsgObject)
  {
    MQeMQMsgObject mqeMsg = (MQeMQMsgObject) msg;
    String replyQMgr = mqeMsg.getReplyToQueueManagerName();
    String replyQueue = mqeMsg.getReplyToQueueName();
    byte [] correlId = mqeMsg.getCorrelationId();
    String msgFormat = mqeMsg.getFormat();
     ...
    byte [] data = mqeMsg.getData();
     ...
  }
```

The data can then be processed by the application. The MQeMQMsgObject returns the data as a `byte array`, and the application must understand the structure of the data within the `byte array`. If the data is required in a more structured format, you can write your own transformer that understands the application data and transforms it into the required format.

Creating an MQSeries style message

To create an MQSeries style message that is understood by the default transformer, create a new MQeMQMsgObject and set the required values for the header fields and data and send the message in the normal way.

To create a new message object invoke the constructor, which has no paramaters.

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
try
{
  MQeMQMsgObject mqeMsg = new MQeMQMsgObject()
  ...
}
```

MQSeries style messages

Set the MQSeries header information in the message using methods of the form `setxxx()` where *xxx* is the name of the header field. For consistency, the names and types of the header fields follow those of the MQSeries Java Client. Any header fields which are not set explicitly assume their MQSeries default value.

The application data is set using the `setData()` method.

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
try
{
  MQeMQMsgObject mqeMsg = new MQeMQMsgObject()
  mqeMsg.setPutApplicationName("myApp");
  mqeMsg.setFormat(...);
  mqeMsg.setData(...);
  MyQM.putMessage(qmgr, queue, mqeMsg, null, 0);
}
```

Before it is passed to `setData()`, the data must be formatted into a `byte array` that the receiving application understands.

Writing a transformer

MQeMQMsgObject is used by the default base transformer in the MQSeries Bridge. An alternative to using this is to write a transformer that uses MQeMsgObject or a subclass of it.

Using the default transformer is a low-cost option when you want a MQSeries Everyplace application to exchange messages with an existing MQSeries application. The main advantage of this approach is its simplicity - the default transformer is already available so the application only has to include code for the MQe application.

This approach becomes less attractive when, for instance:

More than one MQSeries Everyplace application needs to understand the message
: The format of the data in the byte array must be understood by all the applications. With a customized transformer the data format would only have to be understood by the code in the transformer.

Code size in the MQSeries Everyplace application is important
: If you must keep the size of the code for the application to a minimum, all the data formatting code can be put into a transformer. The application can then send and receive the data without having to format it into a byte array. This also removes the need to have the MQeMQMsgObject class on the client device.

Another consideration is that the MQSeries Java Client code is available to the transformer on the MQSeries-bridge to help pack and unpack data into the byte array.

MQSeries-bridge rules

The MQSeries-bridge uses the following rule classes which can be used to alter the behavior of the bridge.

MQeLoadBridgeRule
: This rule class decides which bridges can be loaded when the server starts up.

MQeUndeliveredMessageRule
This rule class decides how to handle an MQSeries message that cannot be put to MQSeries Everyplace

MQeSyncQueuePurgerRule
This rule class decides on the action to take against old unconfirmed MQSeries Everyplace to MQSeries messages

MQeStartupRule
This rule class decides whether an administered object should be started when it is first loaded

These classes are described in more detail in the following sections. As a programmer, you can subclass these rules classes, to create rules to alter the behavior of MQSeries Everyplace, then change your MQSeries Everyplace configuration to use your rule classes instead of the default rule classes.

MQeLoadBridgeRule

This class defines which bridge objects can be loaded when the server starts up. When the server uses the `MQeMQBridge.activate()` method, the bridge loader starts up. The bridge loader reads all entries in the registry and for each name of a bridge in the registry, it asks this rules class whether that bridge name should be loaded or not. The basic MQeLoadBridgeRule class allows all bridges in the registry to be loaded. This is acceptable as long as the registry is used by a single MQSeries Everyplace queue manager.

If the registry is shared by two or more MQSeries Everyplace queue managers they could each try to load the same bridge object, which is not valid. The first server to start up is given access to all the bridges and their queue managers and queues, locking out all subsequent servers. For this reason, it is desirable to select the bridges that should be loaded by each server, by writing a customized version of the MQeLoadBridgeRule. Using a naming convention for the bridges that has some correspondence to the servers that need to load them, simplifies the writing of the customized rule.

The class **examples.mqbridge.rules.ExampleLoadBridgeRule** demonstrates how a naming convention can be applied to bridge objects, and used in conjunction with a LoadBridgeRule, can be used to dictate which bridges may be loaded by the server.

MQeUndeliveredMessageRule

MQeUndeliveredMessageRule

A bridge may have a number of MQSeries transmission queue listener objects defined, and running, each moving a series of messages from an MQSeries transmission queue onto the MQSeries Everyplace network.

When an MQSeries message cannot be delivered to the MQSeries Everyplace network, the transmission queue listener thread consults the UndeliveredMessageRule class in the listener's configuration parameters, by invoking the permit method. The return value from this method dictates what action should be taken.

- If the result is the ″MQeUndeliveredMessageRule.STOP_LISTENER″ value, the listener should stop as a result of this message being undeliverable. The message remains on the transmission queue on the MQSeries system.

- If the result is the ″MQeUndeliveredMessageRule.USE_MQ_REPORT_OPTIONS″ value, the message should either be discarded or moved to the dead letter queue on the MQSeries Everyplace system, depending on the value of the original 'report' field of the original MQSeries message. The name of the MQSeries queue managers' dead letter queue is a configuration parameter on the transmission queue listener bridge object. If this value is returned, and the message report options contain MQRO_DISCARD, then the undelivered message is discarded.
- If the result is an integer, with a value greater than 0, the value returned equates to the number of seconds for which the listener should wait before retrying the MQSeries to MQSeries Everyplace transfer operation.

If the value returned is none of the above, or if the rule throws an exception, then the listener acts as if the STOP_LISTENER result was returned.

The examples.mqbridge.rules.MQeUndeliveredMessageRule class shows the behavior of the default rule used by the MQSeries-bridge configuration: When called, it returns values on successive failures to create the following behavior:

- Waits 5 second between retries for the first minute
- Waits 10 seconds between retries for the second minute
- Waits 60 seconds between retries for the third to ninth minute inclusive
- 'STOP_LISTENER' is applied after retries have failed for 10 minutes

examples.mqbridge.rules.UndeliveredMQMessageToDLQRule is another example class used to tailor the transmission queue listener behavior. The value of 'MQeUndeliveredMessageRule.USE_MQ_REPORT_OPTIONS' is always returned by the `permit()` method.

MQeSyncQueuePurgerRule

The ″sync queue″ is a locally defined queue on the MQSeries queue manager and is used exclusively by the MQSeries-bridge. It is used to assist assured message delivery; for MQSeries Everyplace messages bound for MQSeries it contains one record for each unconfirmed message. Over time, on an unstable system, unconfirmed message records can build up on the sync queue resulting in a degradation of bridge performance.

At an interval specified by the client-connection's sync queue purge interval parameter, the client connection's defined sync queue purger rule class is invoked for each old unconfirmed message record. This rule is asked to return a Boolean 'true' if the supplied message can be deleted or 'false' if it should remain. The administrator can also use this rule to, for example, issue an alert and take appropriate action if a message has not been confirmed after a certain length of time.

See the examples.mqbridge.rules.MQeSyncQueuePurgerRules for more information.

Note: If the sync queue is being used to store the MQSeries transmission queue listener state messages, these messages are not affected by this rule.

MQeStartupRule

When a bridge, proxy, client connection, or listener object is loaded, at server start-up, this rule class is consulted for each administered object in turn to see whether that administered object should be started, or left in the stopped state, and whether the administered objects' children should also be started or not.

The return value from the `MQeStartupRule.permit(...)` method dictates whether the administered object is started or not. Possible return values, and their effects are :

- START_NOTHING - Do not start this administered object. Has the same effect as sending the administered object a ″stop″ administration message.
- START_PARENTS_AND_ME - Start this administered object, and all it's parents. Has the same effect as sending the administered object a ″start″ message, with the ″affect-children″ flag value of false.
- START_PARENTS_AND_ME_AND_CHILDREN - Starts this administered objects, all its' parents, and all it's children. Has the same effect as sending the administered object a ″start″ message, with the ″affect-children″ flag value of true.

As the returned value is controllable programmatically, you could, for example, implement an intelligent rule that only started an MQSeries transmission queue listener if the MQSeries system it needed to connect to was active.

The com.ibm.mqe.mqbridge.MQeStartupRule used in the default configuration for all administered objects is similar to the examples.mqbridge.rules.MQeStartupRule class (for which the source code is provided). These classes always return the START_PARENTS_AND_ME value.

National language support implications

This section uses the diagram in Figure 27 to describe the flow of a message from an MQSeries Everyplace client application over to an MQSeries application.

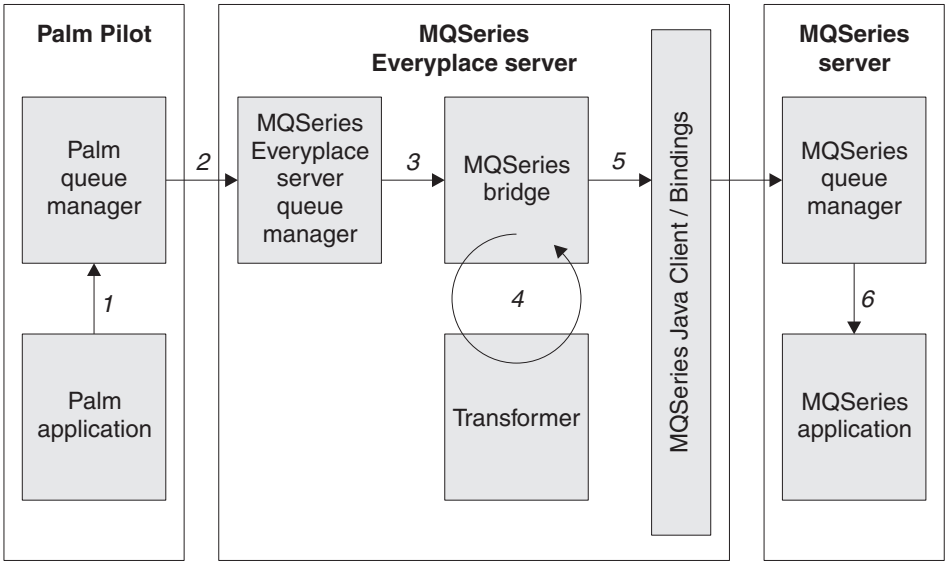

Figure 27. Message flow from MQSeries Everyplace to MQSeries

1. **Client application**
   a. The client application builds an MQSeries Everyplace message object containing the following data:

      A Unicode field
      This string is generated using appropriate libraries available on the client machine (if he is using C/C++).

      A byte field
      This field should never be translated

      An ascii field
      This string has a very limited range of valid characters, conforming to the ASCII standard. The only valid characters are those that are invariant over all ASCII codepages.

   b. The message is put to the palm queue manager. No translation is done during this put.
2. **Client queue manager puts to the server queue manager**
   The message is not translated at all through this step.
3. **MQSeries Everyplace server puts the message onto the MQSeries-bridge queue**
   The message is not translated at all through this step.
4. **MQSeries-bridge passes the MQSeries Everyplace message to the user-written "transformer"**
   The transformer copies information into an MQSeries message. The transformer creates an MQSeries message. The Unicode field in the MQSeries Everyplace message is retrieved using String value = MQemsg.GetUnicode(fieldname) and copied to the MQSeries message using MQmsg.writeChars(value) The byte field in the MQSeries Everyplace message is retrieved using Byte value = MQemsg.getByte(fieldName) and copied to the MQSeries message using

MQmsg.writeByte(value) The ascii field in the MQSeries Everyplace message is retrieved using either MQmsg.writeChars(value) or MQmsg.writeString(value) depending on whether the programmer wishes to create a unicode value (writeChars) or a code-set-dependent value (writeString) in the MQSeries message. If using writeString(), the character set of the string may be set also (it's a member variable of the message). The transformer returns the resultant MQSeries message to the calling MQSeries-bridge code.

5. **The MQSeries-bridge passes the message to MQSeries using the MQSeries java client/bindings classes**

   Unicode values in the MQSeries message are translated from big-endian to little-endian, and vice-versa, as required. Byte values in the MQSeries message are translated from big-endian to little-endian, and vice-versa, as required. A field which was created using writeString() will be translated as the message is put to MQSeries, using conversion routines inside the MQSeries java client code. ASCII data should remain ASCII data regardless of the character set conversions performed. The translations done during this step depend on the code page of the message, the CCSID of the sending MQSeries java client connection, and the CCSID of the receiving MQSeries server connection channel.

6. **The message is ″got″ by an MQSeries application**

   If the message contains a unicode string, the application must deal with that string as a unicode string, or else convert it itself (or using support libraries) into some other format (UTF8 for example). If the message contains a byte string, then the application may use the bytes as-is. (raw data). If the message contains a ″string″, then it is read from the message, and may be converted to a different data format as required by the application (Unicode for example) dependent on the codeset value in the ″characterSet″ header field. Java classes provide this automatically.

Conclusion

If you have an MQSeries Everyplace application, and wish to convey character-related data from MQSeries Everyplace to MQSeries, your choice of method is determined largely by the data you wish to convey:

- **If your data contains characters in the variant ranges of the ASCII character codepages**, (the glpyh for a codepoint changes as you change between the various ASCII codepages) then you can either use `putUnicode` (which will never be subject to translation between codepages), or `putArrayOfByte` (in which case you have to handle the translation between the senders' codepage and the receivers' codepage.

  Note: ***DO NOT USE*** `putAscii()` as the characters in the variant parts of the ASCII codepages are subject to translation.

- **If your data contains only characters in the invariant ranges of the ASCII character codepages**, then you can use `putUnicode` (which will never be subject to translation between codepages) or `putAscii` (which will never be subject to translation between codepages, as all your data lies within the invariant range of the ASCII codepages)

Example files

The following example files are provided to show how to write and use an MQSeries Everyplace program that supports MQSeries-bridge functionality.

bridge examples

examples.mqbridge.awt.AwtMQBridgeServer class

This shows an example of a graphical interface to the underlying examples.mqbridge.queuemanager.MQBridgeServer class.

The MQBridgeServer class source code demonstrates how to add bridge functionality to your MQSeries Everyplace server program, following these guidelines.

To start the bridge enabled server:

1. Instantiate the base MQSeries Everyplace queue manager, and start it running.
2. Instantiate a com.ibm.mqe.mqbridge.MQeMQBridges object, and use it's `activate()` method, passing the same .ini file information as you passed to the base MQSeries Everyplace 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 in-flight bridge operations cleanly, and shuts down all the bridge function.
2. Null-out your reference to the MQeMQBridges object, allowing it to be garbage-collected.
3. Stop and close the base MQSeries Everyplace queue manager.

ExamplesAwtMQBridgeServer.bat

This file provides an example of how to invoke the MQBridgeServer using the Awt server, and 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 bridge functionality.

Chapter 8. Security

This section contains information about the security function provided by MQSeries Everyplace. The different levels of security are described together with typical usage scenarios and usage guidance.

Security features

MQSeries Everyplace provides an integrated set of security features that enable the protection of message data when held locally and when it is being transferred. There are three different categories of security:

Local security
Local security provides protection for MQSeries Everyplace messages while they are held by a local queue manager.

Queue-based security
Queue-based security automatically protects MQSeries Everyplace message data between an initiating queue manager and a target queue, so long as the target queue is defined with an attribute. This protection is independent of whether the target queue is owned by a local or a remote queue manager.

Message-level security
Message-level security provides protection for message data between an initiating and receiving MQSeries Everyplace application.

MQSeries Everyplace local and message-level security are made available to applications. MQSeries Everyplace queue-based security is an internal service.

All three categories protect Message data by the application of an attribute (MQeAttribute or 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 MQSeries Everyplace security. The following sections describe each category of security in detail.

MQSeries Everyplace also provides the following services to assist with security:

Private registry services
MQSeries Everyplace 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 and provides additional services so that when functions that use the entity's private key, (digital signature, and RSA decryption) are invoked, they are supported without the private credentials leaving the PrivateRegistry instance.

© Copyright IBM Corp. 2000

These services are used by queue-based security with and message-level security using MQeTrustAttribute.

Public registry services
MQSeries Everyplace 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
MQSeries Everyplace includes a default *mini-certificate issuance service* that can be configured to issue mini-certificates to a carefully controlled set of entity names.

These services can be used by queue-based and message-level security.

Note: This service is available only in the high security version of MQSeries Everyplace Version 1.0.

Local security

Local security facilitates the protection of MQSeries Everyplace messages (MQeFields descendants) when it is held by a local queue manager. This is achieved by creating an attribute with an appropriate Authenticator, Cryptor and Compressor, setting up an appropriate Key (by providing a password or passphrase seed) and explicitly attaching the Key to the Attribute and the Attribute to the MQeMsgObject. The qualities of the attribute are applied to locally held message data.

The authenticator chosen determines how access to the data is controlled, the choice of cryptor determines the cryptographic strength protecting the data confidentiality, and the choice of compressor determines the size efficiency in storage.

MQSeries Everyplace provides the MQeLocalSecure class to assist with the use of local security, but in all cases it is the responsibility of the local security user to setup an appropriate attribute and provide the password or passphrase secret. 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 (MQeFields object descendent) without using MQeLocalSecure it also needs to save the data after using 'dump' and retrieve the data before using 'restore'. An example of this approach would be an application that wants to use a secure token as backing storage.

The MQSeries Everyplace for Multiplatforms, Programming Reference provides a simple example of the use of MQeLocalSecure. .

Usage scenario

Consider a solution 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 provide a simple method for protecting different customer data held on a given machine (PDA or Laptop for example) using different keys, and possibly different cryptographic strengths.

A simple extension of this scenario could be that protected local data is accessed using a key that is 'pulled' from a queue (with secure attribute) on an area office

MQSeries Everyplace server node. The PDA or Laptop user has to authenticate itself to access the server queue and 'pull' the local key data, but never knows the key that was used.

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 MQeLocalSecure, the following attribute choices are available:

Authenticator
Example NTAuthenticator or UserIdAuthenticator

Cryptor
One of the symmetric cryptors MQeDESCryptor, MQe3DESCryptor, MQeRC4Cryptor, MQeRC6Cryptor or MQeMARSCryptor

Compressor
MQeLZWCompressor or MQeRleCompressor or MQeGZIP compressor

Note: The following cryptors are available only in the high security version of MQSeries Everyplace Version 1.0:

- MQe3DESCryptor
- MQeRC4Cryptor
- MQeRC6Cryptor
- MQeMARSCryptor

Selection criteria

The option to use an authenticator is driven by the 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 or passphrase automatically limits access to those who know this secret.

The choice of cryptor is driven by the strength of protection required, that is, the degree of difficulty that an attacker would face when cryptographically attacking the protected text to get illegal access to the data. Data protected with symmetric ciphers that use 128bit 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.

The option to use a compressor is driven by the need to optimize the size of the protected data. However, the effectiveness of the compressor depends on the content of the data. The MQeRleCompressor performs run length encoding ; that is, the compressor routine compress and/or expand repeated bytes. Hence it is effective in compressing/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 (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 (data patterns).

Usage guide

1. The following program fragments protect an MQeFields object using MQeLocalSecure:

```
try
    {
.../* SIMPLE PROTECT 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 (helper) LocalSecure object                 */
    MQeLocalSecure ls  = new MQeLocalSecure( );
.../* open LocalSecure obj identifying target file and directory*/
    ls.open( ".\\", "TestSecureData.txt" );
.../* use LocalSecure write to encode data and dump to target   */
    trace ( "i: test data in = " + "0123456789abcdef...");
    ls.write( asciiToByte( "0123456789abcdef..." ),
                                        desA, "It_is_a_secret" );
    ...
    }
  catch ( Exception e )
    {
     e.printStackTrace();à             /* 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       = byteToAscii( ls2.read( desA2,
                                          "It_is_a_secret" ) );
.../* show resultsà.                                            */
    trace ( "i: test data out = " + outData);
    ...
    }
  catch ( Exception e )
    {
    e.printStackTrace();               /* show exception          */
    }
```

2. The following program fragment 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 desA  = new MQeAttribute( null, desC, null);
.../* instantiate a base Key object                            */
    MQeKey mylocalkey  = new MQeKey( );
.../* set the base Key object local key                        */
    mylocalkey.setLocalKey( "It_is_a_secret" );
.../* attach the key to the attribute                          */
    desA.setKey( mylocalkey );
.../* activate the attribute                                   */
    desA.activateMaster( null, new MQeFields( ) );
    /* instantiate a Message object                            */
```

```
    MQeMsgObject myMsg = new MQeMsgObject( );
    /* attach the attribute to the message object                */
    myMsg.setAttribute( desA );
    /* add some test message data                                */
    myMsg.putAscii("testdata", "0123456789abcdef....");
    trace ("i: test data in = " + myMsg.getAscii("testdata") );
    /* encode the message                                        */
    byte[] protectedData = myMsg.dump();
    trace ("i: protected test data = byteToAscii(protectedData) );
    }
  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 base Key object                            */
    MQeKey mylocalkey2  = new MQeKey( );
.../* set the base Key object local key                        */
    mylocalkey2.setLocalKey( "It_is_a_secret" );
.../* attach the key to the attribute                          */
    des2A.setKey( mylocalkey2 );
    / instantiate a new msg object                             */
    MQeMsgObject myMsg2 = new MQeMsgObject( );
.../* activate the attribute                                   */
    des2A.activateMaster( null, new MQeFields( ) );
    /* attach the attribute to the message object              */
    myMsg2.setAttribute( des2A );
    /* decode the message data                                 */
    myMsg2.restore( protectedData );
    /* show the unprotected test data                          */
    trace ("i: test data out = " + myMsg2.getAscii("testdata") );
    }
  catch ( Exception e )
    {
    e.printStackTrace();              /* show exception          */
    }
```

Queue-based security

Queue-based security automatically protects MQSeries Everyplace message data between an initiating queue manager and a target queue, so long as the target queue is defined with an attribute. This protection is independent of whether the target queue is owned by a local or a remote queue manager.

A simple example of this is a target queue defined with an attribute that has an NTAuthenticator, an MQe3DESCryptor and an MQeRleCompressor. When such a target queue is accessed (either locally or remotely), using putMessage, getMessage or browseMessages, the queue attribute is automatically applied. In this example the application initiating the access has to satisfy the requirements of the NTAuthenticator before the operation is permitted, and, if permitted, the message data is automatically encoded and decoded using the attribute's MQe3DESCrryptor and MQeRleCompressor. This means that, when the example target queue is remotely accessed, for example using putMessage, queue-based security automatically ensures that the message data is protected at the level defined by the queue attribute, both during transfer between the initiating and remote queue manager and in the target queue backing storage.

Usage scenario

MQSeries Everyplace queue-based security can be used by all solutions that need to protect the confidentiality of message data being transferred between an initiating queue manager and a target queue manager queue.

A typical scenario would be that a solution service is delivered over an open network, for example the internet, where an initiating application makes requests, using the PDA or Laptop resident queue manager, to access a service provided by a server queue manager application.

This can be implemented as follows:

1. The initiating client queue manager application encapsulates the request in an MQSeries Everyplace message
2. putMessage is used to reliably transfer the message to a particular remote server queue manager owned 'XXX_service_request' queue
3. waitForMessage is used , to wait for a reply message to arrive in the local 'XXX_service_reply' queue
4. A server queue manager application is setup to listen for messages on the 'XXX_service_request' queue
5. When a message event occurs, a local getMessage is performed, to get the service request message
6. The request is processed (for example by invocation of a CICS transaction on a backend system)
7. The response(transaction result) is encapsulated in a message
8. putMessage is used to return the response to the remote 'XXX_service_reply' queue owned by the initiating client queue manager.

One way to support this simple example would be to define the following queues:

Owned by the initiating client queue manager (ClientQMgr for example)

- TestClient_HomeServerQ
- XXX_service_reply

While a number of choices exist, setting the TestClient_HomeServerQ TimerInterval option, to 5000 for example, sets a 5sec poll interval and triggers the client queue manager to poll the server queue manager. This poll 'pulls' any messages on the server queue manager's StoreAndForwardQ that have been directed to the client queue manager. Also, before running any ClientQMgr application, the AddQueueManager option must be used to add a reference to the ServerQMgr.

Owned by the server queue manager (ServerQMgr for example)

- TestServer_StoreAndForwardQ
- XXX_service_request

Defining the TestServer_StoreAndForwardQ for use in this scenario requires two steps.

1. Create the queue
2. setAction MQeStoreAndForwardQueueAdminMsg.Action_AddQueueManager, with name "ClientQMgr"

Secure feature choices

When using queue-based security all the choices for attribute are available:

Authenticator
NTAuthenticator or UserIdAuthenticator (or other descendant of examples.attributes.LogonAuthenticator), or MQeWTLSCertAuthenticator

Cryptor
MQeXORCryptor or one of the symmetric crytors MQeDESCryptor, MQe3DESCryptor, MQeRC4Cryptor, MQeRC6Cryptor or MQeMARSCryptor

Compressor
QeLZWCompressor or MQeRleCompressor

Selection criteria

Queue-based security is appropriate for solutions designed to use synchronous queues. In this case, the selection criteria is really concerned with the selection of the (synchronous) queue attribute's authenticator, cryptor and compressor.

The option to use an Authenticator is driven by the need to provide additional controls to prevent access to the local data by unauthorized users. This is equally relevant when the queue data is accessed locally or remotely.

Using a descendant of LogonAuthenticator (NTAuthenticator or UserIdAuthenticator), when the attribute is activated, for example when an application is performing a putMessage, getMessage or browseMessages of data on the queue, the requirements of the authenticator have to be satisfied before the operation is permitted. In the queue-based "Usage scenario" on page 162, if the XXX_service_request queue is defined with an attribute including the NTAuthenticator, then access to the server XXX_service_request queue (for example when attempting to putMessage requests to this queue from a client) queue manager), is restricted to the set of users defined as valid NT users in the target server's domain. The NTAuthenticator is provided as an example, enabling descendents that enable a finer granularity of control to smaller sets of users to be easily created.

Using MQeWTLSCertAuthenticator ensures that all remote accesses to a queue protected with an attribute using this authenticator have completed mutual authentication before the operation can be executed. The mutual authentication of the mini-certificates exchanged consists of each participant validating the mini-certificate it receives. This validation checks the mini-certificate received is a valid signed entity, signed by the same mini-certificate server as the requestor's own mini-certificate, and that it is valid with respect to date, that is the current date is not prior to its from-date or after its to-date. An administration option enables the solution creator to choose whether a target queue manager queue has its own credentials (that it is an authenticatable entity in its own right, with its own mini-certificate and associated private key) or shares the credentials of its owning queue manager. In the queue-based "Usage scenario" on page 162, if the XXX_service_request queue is defined with an attribute containing the MQeWTLSCertAuthenticator, then access to the server XXX_service_request queue, for example when the initiating client queue manager application performs a remote putMessage, depends on the credentials of the initiating client queue manager and the target XXX_service_request queue being successfully mutually authenticated.

The choice of cryptor is driven by the strength of protection required, that is, the degree of difficulty that an attacker would face when cryptographically attacking

the protected data to get illegal access. Data protected with symmetric ciphers which use 128bit keys is acknowledged as being more difficult to attack than data protected using ciphers that use shorter keys. But in addition to cryptographic strength. the selection of a cipher may also be driven by many other factors. An example of this is some financial solutions require the use of triple DES in order to get audit approval.

The option to use a compressor is driven by the need to optimize the size of the protected data. However, the effectiveness of the compressor depends on the content of the data. The MQeRleCompressor performs run length encoding ; that is, the compressor routine compress and/or expand repeated bytes. Hence it is effective in compressing/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 (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 (data patterns).

Note: The following cryptors and authenticators are available only in the high security version of MQSeries Everyplace Version 1.0:

- MQe3DESCryptor
- MQeRC4Cryptor
- MQeRC6Cryptor
- MQeMARSCryptor
- MQeWTLSCertAuthenticator

Usage guide

The following code fragments provide an example of how to create queue manager instances and define the queues identified for the queue-based scenario described in “Usage scenario” on page 162. Fragments for the client queue manager initiating application and server queue manager AppRunList started application are also provided.

Using SimpleCreateQM to create ClientQMgr and ServerQMgr instances

SimpleCreateQM assists users to create queue manager instances that have private registries. The class uses parameters found in the Registry Section of `MQePrivateClient1.ini` and `MQePrivateServer1.ini.`

The particular instances can be created as follows:

1. Reset the private registry related parameters in the registry section of `MQePrivateClient1.ini` and `MQePrivateServer1.ini` from their defaults to a desired setting:

```
(ascii)LocalRegType=PrivateRegistry
(ascii)DirName=.\\MQeNode_PrivateRegistry
(ascii)PIN=12345678
  < change PIN from '12345678' to the PIN to be provided subsequently at
     queue manager  start-up time to enable the queue manager to access its
    own private registry >
```

   Include the next three keywords (CertReqPIN, KeyRingPassword and CAIPAddrPort only if MQeWTLSCertAuthenticator is to be used:

```
(ascii)CertReqPIN=12345678
  < change CertReqPIN from '12345678' to a new value that matches the value
     set  value defined by Mini Certertificate Server's Administrator when the
```

```
      queuemanager instance is defined >
  (ascii)KeyRingPassword=It_is_a_secret
    < change the KeyRingPassword  from 'It_is_a_secret'  to the password that
      to be subsequently provided at queuemanager start-up time to enable
      the queuemanager instance to access its protected private credentials
      within its Private Registry. >
  (ascii)CAIPAddrPort=9.20.X.YYY:8081
    < change this to the IP address and port of the solution's
       MiniCertificateServer
```

2. If the last three keywords are supplied auto-registration is be triggered, so, before adding the queue manager instances it is necessary to start the MiniCerificateServerGUI, and, using 'Administration' mode, to define the queue manager instances (ClientQMgr and ServerQMgr) as valid authenticatable entities with their certificate request PIN set to the same value as that defined in the registry section `CertReqPIN=` line in the `MQePrivateClient1.ini` and `MQePrivateServer1.ini` files in the previous step.
3. Start a MiniCertificateServerGUI instance and select 'Server' mode.
4. Run the `TestCreate` program (shown in the following code fragment) to create the queue manager instances.

```
package test;
import com.ibm.mqe.*;
import examples.install.*;
public class TestCreate extends MQe
  {
  public void createQMs( )
    {
    /* start traceã                                                    */
    try{
       MQeTraceInterface trace =
        (MQeTraceInterface) MQe.loader.loadObject(
                                          "examples.awt.AwtMQeTrace" );
       trace.activate( "TestCreate...", null );
       }
    catch(Exception e) {e.printStackTrace(); }
    try{
       String INI_FileName   = ".\MQePrivateClient1.ini";
       String QueueDir       = ".\ClientQMgr\Queues\";
       SimpleCreateQM c_QMgr = new SimpleCreateQM();
       if ( c_QMgr.createQMgr(INI_FileName, QueueDir) )
         trace ( ">>>> ClientQMgr created OK...");
       else
         trace (">>>> error creating ClientQMgr...");
       INI_FileName          = ".\MQePrivateServer1.ini";
       QueueDir              = ".\ServerQMgr\Queues\";
       SimpleCreateQM s_QMgr = new SimpleCreateQM();
       if ( s_QMgr.createQMgr(INI_FileName, QueueDir) )
         trace ( ">>>> ServerQMgr created OK...");
       else
         trace (">>>> error creating ServerQMgr...");
       }
     catch (Exception e)
       {
       trace (">>>> SimpleCreateQM eception = "+ e.getMessage( ) );
       e.printStackTrace();
       }
    }
  public static void main(String args[])
    {
    TestCreate testc = new TestCreate( );
    testc.createQMs( );
    }
  }
```

Defining the queues identified for the queue-based scenario described above

There are several ways add queue definitions to a queue manager instance. The method described here starts the queue manager instance locally, adds the new queue definitions by creating the relevant administration messages and sending them to the queue manager's own administration queue, and then waits for confirmation of success in an AdminReply queue.

ClientQMgr queues -adding TestClient_HomeServerQ:

Start the ClientQMgr locally using the MQePrivateClient class, (using a different version, MQePrivateClient2.ini, that deliberately does not hold hard coded values for PIN, KeyRingPassword and CertReqPIN) then create and use an administration messages to add the queue and set the poll interval.

```
{
try{
   /* start ClientQMgrà                                              */
   String QMgrName       = "ClientQMgr";
   String QName          = "TestClient_HomeServerQ"
   MQeAttribute qattr    = new MQeAttribute(null,
                                    new MQe3DESCryptor, null);
   String FileDesc       = "MsgLog:.";
   MQePrivateClient newC = new MQePrivateClient(
                            ".//MQePrivateClient2.ini",
                            "12345678",         /* or new PIN       */
                            "It_is_a_secret",   /* or new KeyRingPwd*/
                             null);
   MQeQueueManager newQM =  newC.queueManager;
   /* create and use Admin msg to add HomeServerQà                   */
   MQeHomeServerQueueAdminMsg msg  =
     new MQeHomeServerQueueAdminMsg("ServerQMgr",
                                    "ServerTestQ_StoreAndForward");
   MQeFields parms       = new MQeFields( );
   parms.putLong( MQeHomeServerQueueAdmin.Queue_QTmerInterval, 5000 );
   msg.setTargetQMgr( QMgrName );
   msg.setName( QMgrName, QName );
   msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
   msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
   msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
   msg.putArrayOfByte( MQe.Msg_CorrelID,
            Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
   MQeFields msgTest = new MQeFields( );
   msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                           msg.getArrayOfByte( MQe.Msg_CorrelID ) );
   parms.putAscii( msg.Queue_QMgrName, "ServerQMgr" );
   parms.putAscii( msg.Queue_FileDesc, FileDesc );

   if ( qattr.getAuthenticator( ) != null ) /*add qattr auth details*/
     {
     parms.putAscii( msg.Queue_Authenticator,
                                  qattr.getAuthenticator( ).type( ) );
     if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
       {
       parms.putAscii( msg.Queue_AttrRule,
                                    "examples.rules.AttributeRule" );
       parms.putByte( msg.Queue_TargetRegistry,
                                           msg.Queue_RegistryQueue );
       }
     }
   if ( qattr.getCryptor( ) != null )
     {
     parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
     if (! parms.contains( msg.Queue_AttrRule ) )
```

```
          parms.putAscii( msg.Queue_AttrRule,
                                      "examples.rules.AttributeRule" );
        }
      if ( qattr.getCompressor( ) != null )
        parms.putAscii( msg.Queue_Compressor,
                               qattr.getCompressor( ).type( ) );
      parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
      msg.create( parms );
      trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
      /* use Admin msg to add HomeServerQà                              */
      newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
      MQeAdminMsg respMsg      = null;
      trace(">>> Waiting for a response to create Admin Msg...");
      respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                               "AdminReplyQ", msgTest, null, 0, 3000);
      trace(">>> Admin Msg processed OK...");
      /* process Admin msg response                 à                   */
      if ( respMsg == null )
        trace ( "i: create Queue failed, no response message received" );
      else
        {
        if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
          trace( "i: create Queue added queue OK..." );
        else
          trace( "i: create Queue failed: " + respMsg.getReason( ) );
        }
      newQM.close();
      }
    catch ( Exception e )
      {
      trace (">>>> add HomeServerQ exception = "+ e.getMessage( ) );
      e.printStackTrace();
      }
     }
```

ClientQMgr queues -adding XXX_service_reply queue:

Start the ClientQMgr locally using the MQePrivateClient class, (using a different version, MQePrivateClient2.ini, that deliberately does not hold hard coded values for PIN, KeyRingPassword and CertReqPIN) then create and use an administration messages to add the queue.

```
{
    try{
       /* start ClientQMgrà                                           */
       String QMgrName       = "ClientQMgr";
       String QName          = "XXX_service_reply"
       MQeAttribute qattr    = new MQeAttribute(null,
                                          new MQe3DESCryptor, null);
       String FileDesc       = "MsgLog:.";
       MQePrivateClient newC = new MQePrivateClient(
                                ".//MQePrivateClient2.ini",
                                "12345678",         /* or new PIN       */
                                "It_is_a_secret",   /* or new KeyRingPwd*/
                                 null);
       MQeQueueManager newQM =  newC.queueManager;
       /* create and use Admin msg to add XXX_service_reply queue       */
       MQeQueueAdminMsg msg  = new MQeQueueAdminMsg( );
       MQeFields parms       = new MQeFields( );
       msg.setTargetQMgr( QMgrName );
       msg.setName( QMgrName, QName );
       msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
       msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
       msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
       msg.putArrayOfByte( MQe.Msg_CorrelID,
                Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
```

```
    MQeFields msgTest = new MQeFields( );
    msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                            msg.getArrayOfByte( MQe.Msg_CorrelID ) );
    parms.putAscii( msg.Queue_QMgrName, "ServerQMgr" );
    parms.putAscii( msg.Queue_FileDesc, FileDesc );
     if ( qattr.getAuthenticator( ) != null )
      {
      parms.putAscii( msg.Queue_Authenticator,
                                  qattr.getAuthenticator( ).type( ) );
      if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
        {
        parms.putAscii( msg.Queue_AttrRule,
                                    "examples.rules.AttributeRule" );
        parms.putByte( msg.Queue_TargetRegistry,
                                           msg.Queue_RegistryQueue );
        }
      }
    if ( qattr.getCryptor( ) != null )
      {
      parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
      if (! parms.contains( msg.Queue_AttrRule ) )
        parms.putAscii( msg.Queue_AttrRule,
                                    "examples.rules.AttributeRule" );
      }
    if ( qattr.getCompressor( ) != null )
      parms.putAscii( msg.Queue_Compressor,
                                qattr.getCompressor( ).type( ) );
    parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
    msg.create( parms );
    trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
    /* use Admin msg to add queue     ã                              */
    newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
    MQeAdminMsg respMsg      = null;
    trace(">>> Waiting for a response to create Admin Msg...");
    respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                                "AdminReplyQ", msgTest, null, 0, 3000);
    trace(">>> Admin Msg processed OK...");
    /* process Admin msg response                 ã                  */
    if ( respMsg == null )
      trace ( "i: create Queue failed, no response message received" );
    else
      {
      if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
        trace( "i: create Queue added queue OK..." );
      else
        trace( "i: create Queue failed: " + respMsg.getReason( ) );
      }
    newQM.close();
    }
  catch ( Exception e )
    {
    trace (" >>> add XXX_service_reply Q excep = "+ e.getMessage( ) );
    e.printStackTrace();
    }
  }
```

ServerQMgr queues -adding TestServer_StoreAndForwardQ:

Start the ServerQMgr locally using the MQePrivateClient class, (using a different version, `MQePrivateServer2.ini`, that deliberately does not hold hard coded values for PIN, KeyRingPassword and CertReqPIN), create and use an administration messages to add the queue, and then add a remote queue manager reference.

```
{
    try{
       /* start ServerQMgr, locally                                  */
       String QMgrName       = "ServerQMgr";
```

```
String QName          = "TestServer_StoreAndForwardQ"
MQeAttribute qattr    = new MQeAttribute(null,
                                    new MQe3DESCryptor, null);
String FileDesc       = "MsgLog:.";
MQePrivateClient newC = new MQePrivateClient(
                         ".//MQePrivateServer2.ini",
                         "12345678",         /* or new PIN       */
                         "It_is_a_secret",   /* or new KeyRingPwd*/
                          null);
MQeQueueManager newQM =  newC.queueManager;
/* create and use Admin msg to add StoreAndForwardQ              */
MQeStoreAndForwardQueueAdminMsg( ) msg  =
                            new MQeStoreAndForwardQueueAdminMsg( );
MQeFields parms       = new MQeFields( );
msg.setTargetQMgr( QMgrName );
msg.setName( QMgrName, QName );
msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
msg.putArrayOfByte( MQe.Msg_CorrelID,
        Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
MQeFields msgTest = new MQeFields( );
msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                      msg.getArrayOfByte( MQe.Msg_CorrelID ) );
parms.putAscii( msg.Queue_QMgrName, QMgrName );
parms.putAscii( msg.Queue_FileDesc, FileDesc );
if ( qattr.getAuthenticator( ) != null )
  {
  parms.putAscii( msg.Queue_Authenticator,
                               qattr.getAuthenticator( ).type( ) );
  if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
    {
    parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
    parms.putByte( msg.Queue_TargetRegistry,
                                        msg.Queue_RegistryQueue );
    }
  }
if ( qattr.getCryptor( ) != null )
  {
  parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
  if (! parms.contains( msg.Queue_AttrRule ) )
    parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
  }
if ( qattr.getCompressor( ) != null )
  parms.putAscii( msg.Queue_Compressor,
                             qattr.getCompressor( ).type( ) );
parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
msg.create( parms );
trace(" >>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
/* use Admin msg to add queue     à                              */
newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
MQeAdminMsg respMsg      = null;
trace(" >>> Waiting for a response to create Admin Msg...");
respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                             "AdminReplyQ", msgTest, null, 0, 3000);
trace(" >>> Admin Msg processed OK...");
/* process Admin msg response                  à                 */
if ( respMsg == null )
  trace ( "i: create Queue failed, no response message received" );
else
  {
  if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
    trace( "i: create Queue added queue OK..." );
  else
    trace( "i: create Queue failed: " + respMsg.getReason( ) );
```

```
      }
    /* use Admin msg to StoreAndForwardQ AddQueueManager reference  */
    msg = new MQeStoreAndForwardQueueAdminMsg( );
    msg.addQueueManager( "ClientQMgr" );
    parms = new MQeFields( );
    msg.setTargetQMgr( QMgrName );
    msg.setName( QMgrName, QName );
    msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
    msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
    msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
    msg.putArrayOfByte( MQe.Msg_CorrelID,
              Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
    MQeFields msgTest = new MQeFields( );
    msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                            msg.getArrayOfByte( MQe.Msg_CorrelID ) );
    parms.putAscii( msg.Queue_QMgrName, QMgrName );
    parms.putAscii( msg.Queue_FileDesc, FileDesc );
    msg.setAction(
            MQeStoreAndForwardQueueAdminMsg.Action_AddQueueManager );
    trace(" >>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
    newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
    MQeAdminMsg respMsg = null;
    trace(" >>> Waiting for a response to update Admin Msg...");
    respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                             "AdminReplyQ", msgTest, null, 0, 3000);
    trace(" >>> Admin Msg processed OK...");
    /* process Admin msg response                à                  */
    if ( respMsg == null )
      trace ( "i: create Queue failed, no response message received" );
    else
      {
      if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
        trace( "i: create Queue added queue OK..." );
      else
        trace( "i: create Queue failed: " + respMsg.getReason( ) );
      }
    trace(" >>> StoreAndForwardQ AddQueueManager reference OK..." );
    newQM.close();
    }
  catch ( Exception e )
    {
    trace (" >>> add StoreAndForwardQ exception = "+ e.getMessage( ) );
    e.printStackTrace();
    }
```

ServerQMgr queues -adding XXX_service_request queue:

Start the ServerQMgr locally using the MQePrivateClient class, (using a different version, MQePrivateServer2.ini, that deliberately does not hold hard coded values for PIN, KeyRingPassword and CertReqPIN) then create and use an administration messages to add the queue.

```
{
    try{
       /* start ServerQMgrà                                          */
       String QMgrName       = "ServerQMgr";
       String QName          = "XXX_service_request"
       MQeAttribute qattr    = new MQeAttribute(null,
                                        new MQe3DESCryptor, null);
       String FileDesc       = "MsgLog:.";
       MQePrivateClient newC = new MQePrivateClient(
                               ".//MQePrivateServer2.ini",
                               "12345678",         /* or new PIN       */
                               "It_is_a_secret",   /* or new KeyRingPwd*/
                                null);
       MQeQueueManager newQM =  newC.queueManager;
```

```
   /* create and use Admin msg to add XXX_service_request queue     */
   MQeQueueAdminMsg msg  = new MQeQueueAdminMsg( );
   MQeFields parms       = new MQeFields( );
   msg.setTargetQMgr( QMgrName );
   msg.setName( QMgrName, QName );
   msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
   msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
   msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
   msg.putArrayOfByte( MQe.Msg_CorrelID,
            Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
   MQeFields msgTest = new MQeFields( );
   msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                           msg.getArrayOfByte( MQe.Msg_CorrelID ) );
   parms.putAscii( msg.Queue_QMgrName, QMgrName );
   parms.putAscii( msg.Queue_FileDesc, FileDesc );


   if ( qattr.getAuthenticator( ) != null ) /*add qattr auth details*/
     {
     parms.putAscii( msg.Queue_Authenticator,
                                  qattr.getAuthenticator( ).type( ) );
     if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
       {
       parms.putAscii( msg.Queue_AttrRule,
                                    "examples.rules.AttributeRule" );
       parms.putByte( msg.Queue_TargetRegistry,
                                           msg.Queue_RegistryQueue );
       }
     }
   if ( qattr.getCryptor( ) != null )
     {
     parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
     if (! parms.contains( msg.Queue_AttrRule ) )
       parms.putAscii( msg.Queue_AttrRule,
                                    "examples.rules.AttributeRule" );
     }
   if ( qattr.getCompressor( ) != null )
     parms.putAscii( msg.Queue_Compressor,
                               qattr.getCompressor( ).type( ) );
   parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
   msg.create( parms );
   trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
   /* use Admin msg to add XXX_service_request queue                */
   newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
   MQeAdminMsg respMsg      = null;
   trace(">>> Waiting for a response to create Admin Msg...");
   respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                               "AdminReplyQ", msgTest, null, 0, 3000);
   trace(">>> Admin Msg processed OK...");
   /* process Admin msg response                 ã                  */
   if ( respMsg == null )
     trace ( "i: create Queue failed, no response message received" );
   else
     {
     if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
       trace( "i: create Queue added queue OK..." );
     else
       trace( "i: create Queue failed: " + respMsg.getReason( ) );
     }
   newQM.close();
   }
catch ( Exception e )
   {
   trace (" >>> add XXX_service_request excep = "+ e.getMessage( ) );
```

```
      e.printStackTrace();
      }
    }
```

Server queue manager AppRunList started application.:

This section provides an example extension to `MQePrivateServer2.ini` showing how to add an AppRunList application that is automatically started when the ServerQMgr starts. It also provides an example `TestService` application.

Example MQePrivateServer2.ini

```
MQePrivateServer2.ini - with AppRunList extensionà
[Alias]
(ascii)EventLog=examples.log.LogToDiskFile
(ascii)Network=com.ibm.mqe.adapters.MqeTcpipHttpAdapter
(ascii)QueueManager=com.ibm.mqe.MqeQueueManager
(ascii)Trace=examples.awt.AwtMQeTrace
(ascii)MsgLog=com.ibm.mqe.adapters.MqeDiskFieldsAdapter
(ascii)FileRegistry=com.ibm.mqe.registry.MqeFileSession
(ascii)PrivateRegistry=com.ibm.mqe.registry.MqePrivateSession
(ascii)ChannelAttrRules=examples.rules.AttributeRule
(ascii)AttributeKey_1=com.ibm.mqe.MQeKey
(ascii)AttributeKey_2=com.ibm.mqe.attributes.MqeSharedKey
[ChannelManager]
(int)MaxChannels=0
[Listener]
(ascii)Listen=Network::8081
(ascii)Network=Network:
(int)TimeInterval=300
[QueueManager]
(ascii)Name=ServerQMgr
(ascii)QueueStore=MsgLog:.\MQeNode_PrivateRegistry
[Registry]
(ascii)LocalRegType=PrivateRegistry
(ascii)DirName=.\\MQeNode_PrivateRegistry
(ascii)PIN=not set
(ascii)CertReqPIN=not set
(ascii)KeyRingPassword=not set
(ascii)CAIPAddrPort=9.20.X.YYY:8081
[AppRunList]
(ascii)App1=test.TestService
```

Example Server TestSevice application

```
package test;
import com.ibm.mqe.*;
import com.ibm.mqe.attributes.*;
import java.util.*;
public class TestService extends MQe
   implements MQeRunListInterface, MQeMessageListenerInterface, Runnable
  {
  protected Thread applicationThread = null;
  protected MQeQueueManager thisQMgr = null;

  /* constructor                                                    */
  public TestService( ) throws Exception
    {
    }

  /* activate method                                                */
  public Object activate( Object    owner,
                          Hashtable loadTable,
                          MQeFields setupData ) throws Exception
```

```
  {
  System.out.println(" TestService, activate, owner objref = " + owner);
  thisQMgr           = (MQeQueueManager)owner;  /* save QMgr objref   */
  applicationThread  = new Thread(
               this, "applicationThread" ); /* create svr app thread */
  System.out.println(" TestService, activate no of active threads = " +
                                                 Thread.activeCount( ) );
  Thread t[] = new Thread[Thread.activeCount( )];
  int i = Thread.enumerate( t );
  for ( int j = 0; j < i; j++ )              /* look at svr threads   */
    System.out.println("TestService activate, active thread name = "
                                                      + t[j].getName( ) );
  applicationThread.start( );                /* start appl'n Thread.  */
  return this;
  }

/* run method                                                          */
public void run( )
  {
  System.out.println("TestService, Run...");
  /* add listener for XXX_service_request queue                       */
  try {
      thisQMgr.addMessageListener( this, "XXX_service_request",
                                                    new MQeFields( ) );
      }
  catch( Exception e)
      {
      e.printStackTrace( );
      }
  }


/* MessageArrived event handler                                        */
/* MsgArrived event is generated when a message arrives on a queue     */
public void messageArrived( MQeMessageEvent msgEvent )
{
try {
    System.out.println(" TestService, msgEvent, messageArrived ");
    System.out.println(" TestService, msgEvent getQueueManagerName = " +
                                     msgEvent.getQueueManagerName() );
    System.out.println(" TestService, msgEvent getQueueName = " +
                                           msgEvent.getQueueName( ) );
    /* get XXX service request message                                 */
    MQeMsgObject reqmsg = thisQMgr.getMessage(
                            msgEvent.getQueueManagerName( ),
                            msgEvent.getQueueName( ),
                            msgEvent.getMsgFields( ),
                            null,
                            0);
    /* process service request here                                    */
    String reqdata   = reqmsg.getAscii("XXX_service_request_data");
    String replydata = reqdata + "_reply";
    /* build XXX_service reply message here                            */
    MQeMsgObject replymsg = new MQeMsgObject( );
    replymsg. putArrayOfByte( MQe.Msg_CorrelID,
                             reqmsg.getArrayOfByte(MQe.Msg_CorrelID ) );
    replymsg.putAscii("XXX_service_reply_data", replydata );
    System.out.println(" TestService, msgEvent putting service reply " +
                          "to ClientQMgr XXX_service_reply queue");
    /* put reply to ClientQMgr XXX_service reply queue                 */
    thisQMgr.putMessage( "ClientQMgr", "XXX_service_reply",
                                                    replymsg, null, 1 );
    }
catch( Exception e )
    {
    e.printStackTrace( );
    }
```

```
  }
  /* finalize method                                                  */
  protected void finalize()
    {
    System.out.println("TestService, finalize...");
    applicationThread.stop( );
    applicationThread.destroy( );
    }
```

Client queue manager application initiating XXX_service_request.:

The example queue-based security scenario in "Usage scenario" on page 162 describes a client queue manager application that initiates XXX_service_request messages by encapsulating the request in a MQeMsgObject and using putMessage to reliably deliver the request to the server queue manager's XXX_sevice_request queue. It then waits for the reply to the service request by using waitForReply on its own XXX_service_reply queue.

In the scenario, the TestService application on the server processes the service request by using getMessage to get the service request from the XXX_service_request queue, processes the request (for example by invocation of a backend transaction), builds the reply MQeMsgObject, and uses the server queue manager putMessage to return the reply to the (remote) initiating Client queue manager.

The server queue manager internally puts the message onto its TestServer_StoreAndForwardQ. The client queue manager pulls the message from the TestServer_StoreAndForwardQ and receives it in its ClientTest_HomeServerQ before putting it on the intended target XXX_service_reply queue.

The client application below provides a simple example of invoking a service request and processing the resulting reply.

```
package test;
import com.ibm.mqe.*;
import examples.queuemanager.*;
 public class UseTestService extends MQe
  {
  protected MQeQueueManager thisQMgr = null;
  /* serviceRequest method                                            */
  public void serviceRequest( )
    {
    /* start traceà                                                   */
    try{
       MQeTraceInterface trace =
        (MQeTraceInterface) MQe.loader.loadObject(
                                        "examples.awt.AwtMQeTrace" );
       trace.activate( "UseTestService...", null );
       }
    catch(Exception e) {e.printStackTrace();
    /* start and use Client queuemanager to put request & process reply */
    try {
        /* start Client queue manager                                 */
        MQePrivateClient newC = new MQePrivateClient(
                                     ".//MQePrivateClient2.ini",
                                     "12345678",
                                     "It_is_a_secret",
                                     null );
        MQeQueueManager newQM =  newC.queueManager;
        /* build svc request and use putMessage to put it to server   */
        MQeMsgObject msgreq   = new MQeMsgObject( );
        long thisReq_CorrelID = newQM.uniqueValue();
```

```
      msgreq.putArrayOfByte( MQe.Msg_CorrelID,
                                        longToByte( thisReq_CorrelID) );
      String reqdata        = "0123456789abcdef";
      msgreq.putArrayOfByte("XXX_service_request_data",
                                              asciiToByte(reqdata) );
      newQM.putMessage("ServerQMgr","XXX_service_reqest",msgreq,null,1);
      trace( " >>> request put to ClientQMgr,XXX_service_request q OK");
      /* field and process reply to service request                  */
      trace( " >>> waiting for reply message...");
      MQeFields msgreq_filter = new MQeFields();
      msgreq_filter.putArrayOfByte( MQe.Msg_CorrelID,
                                        longToByte( thisReq_CorrelID) );
      MQeMsgObject msgreply  = newQM.waitForMessage( newQM.getName( ),
                   "XXX_service_reply", msgreq_filter, null, 0, 3000 );
      trace(" >>> service request reply = " +
        byteToAscii(msgreply.getArrayOfByte("XXX_service_reply_data")));
      }
    catch(Exception e2) { e2.printStackTrace();}
    }
  }
  public static void main(String args[])
    {
    UseTestService testsvc = new UseTestService( );
    testsvc.serviceRequest();
    }
  }
```

Queue-based security and triggering auto-registration

When a queue manager accesses a remote queue or any local queue that is defined with an attribute including the MQeWTLSCertAuthenticator, then the queue manager and queues are authenticatable entities and require their own credentials.

A queue manager's credentials are created by triggering auto-registration. The simplest way of triggering auto-registration is to include the relevant keywords in the registry section of the `ini` file used when the queue manager is created: The keywords needed in the registry section of the `ini` file are:

```
(ascii)CertReqPIN=12345678
  < change CertReqPIN the default '12345678' to a new value that matches the value
     set  value defined by Mini Certertificate Server's Administrator when the
     QueueManager instance is defined >
(ascii)KeyRingPassword=It_is_a_secret
  < change the default KeyRingPassword  from 'It_is_a_secret'  to the password that
     is to be subsequently provided at QueueManager start-up time to enable
     the QueueManager instance to access its protected private credentials
     within its Private Registry. >
(ascii)CAIPAddrPort=9.20.X.YYY:8081
  < change this to the IP address and port of the solution's MiniCertificateServer
```

The credentials of queues (with an attribute including MQeWTLSCertAuthenticator) are also created by triggering auto-registration. This happens automatically when an administration message adding the queue is processed providing that:

- The owning queue manager has already auto-registered, and been started with parameters necessary to access its own credentials and the solutions's mini-certificate server
- The owning queue manager name and queue name have been predefined by the mini-certificate server administrator, with the mini-certificate request PIN set to the same value as the CertReqPIN value used to start the owning queue manager
- The mini-certificate server is available, started, and is in 'Server' mode

When adding a queue (with an attribute including MQeWTLSCertAuthenticator) the queue can have its own credentials or it can share its owning queue manager's credentials. This choice is determined when the 'create queue' administration message is constructed. The following code fragment shows the relevant parameters and their meaning.

ServerQMgr queues -adding ServerTestQWTLS2:

The following code fragment:

- Assumes that the mini-certificate server administrator has added ServerQMgr+ServerTesTQWTLS2 with Certificate Request PIN = 12345678, and has started the mini-certificate server in 'Server' mode
- Starts the ServerQMgr locally using the MQePrivateClient class, (using the different version, MQePrivateServer2.ini, that deliberately does not hold hard coded values for PIN, KeyRingPassword and CertReqPIN) then create and use an administration message to add the ServerTestQWTLS2 queue

```
{
    try{
       /* start ServerQMgrà                                              */
       String QMgrName       = "ServerQMgr";
       String QName          = "ServerTestQWTLS2"
       MQeAttribute qattr    = new MQeAttribute(
               new MQeWTLSCertAuthenticator(), new MQe3DESCryptor, null);
       String FileDesc       = "MsgLog:.";
       MQePrivateClient newC = new MQePrivateClient(
                               ".//MQePrivateServer2.ini",
                               "12345678",         /* or new PIN       */
                               "It_is_a_secret",   /* or new KeyRingPwd*/
                                null);
       MQeQueueManager newQM =  newC.queueManager;
       /* create and use Admin msg to add ServerTestQWTLS2 queue        */
       MQeQueueAdminMsg msg  = new MQeQueueAdminMsg( );
       MQeFields parms       = new MQeFields( );
       msg.setTargetQMgr( QMgrName );
       msg.setName( QMgrName, QName );
       msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
       msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
       msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
       msg.putArrayOfByte( MQe.Msg_CorrelID,
                Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
       MQeFields msgTest = new MQeFields( );
       msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                              msg.getArrayOfByte( MQe.Msg_CorrelID ) );
       parms.putAscii( msg.Queue_QMgrName, QMgrName );
       parms.putAscii( msg.Queue_FileDesc, FileDesc );
       if ( qattr.getAuthenticator( ) != null ) /*add qattr auth details*/
         {
         parms.putAscii( msg.Queue_Authenticator,
                                    qattr.getAuthenticator( ).type( ) );
         if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
           {
           parms.putAscii( msg.Queue_AttrRule,
                                      "examples.rules.AttributeRule" );
           /* for the Queue to have its own credentials                 */
           parms.putByte( msg.Queue_TargetRegistry,
                                           msg.Queue_RegistryQueue );
           /* for the Queue to share its host QMgr's cresdentials       */
          // parms.putByte( msg.Queue_TargetRegistry,
          //                                 msg.Queue_RegistryQMgr );
           }
         }
       if ( qattr.getCryptor( ) != null )
         {
```

```
      parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
      if (! parms.contains( msg.Queue_AttrRule ) )
        parms.putAscii( msg.Queue_AttrRule,
                                    "examples.rules.AttributeRule" );
      }
    if ( qattr.getCompressor( ) != null )
      parms.putAscii( msg.Queue_Compressor,
                              qattr.getCompressor( ).type( ) );
    parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
    msg.create( parms );
    trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
    /* use Admin msg to add ServerTestQWTLS2                          */
    newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
    MQeAdminMsg respMsg      = null;
    trace(">>> Waiting for a response to create Admin Msg...");
    respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                              "AdminReplyQ", msgTest, null, 0, 3000);
    trace(">>> Admin Msg processed OK...");
    /* process Admin msg response                 à                   */
    if ( respMsg == null )
      trace ( "i: create Queue failed, no response message received" );
    else
      {
      if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
        trace( "i: create Queue added queue OK..." );
      else
        trace( "i: create Queue failed: " + respMsg.getReason( ) );
      }
    newQM.close();
    }
  catch ( Exception e ) { e.printStackTrace(); }
  }
```

Queue-based security, starting queue managers with private registries

Whenever a queue manager and any of its queues are authenticatable entities, that is, have their own credentials, then, in order to access these credentials, the appropriate parameters are needed when the queue manager is started.

While hard coding these parameters in the registry section of the appropriate `ini` file is a convenient mechanism during solution development, it is inappropriate for a production system. Whenever possible, these parameters should be collected interactively and used to start a queue manager instance without storing them in a file.

An example of starting an MQSeries Everyplace client queue manager using the MQePrivateClient class, and passing the parameters (instead of hard coding them in keywords of the MQePrivateClient2.ini file) is found in the example “ClientQMgr queues -adding XXX_service_reply queue” on page 167.

Queue-based security - channel reuse

When protecting data between an initiating queue manager and target queues (owned by the same remote queue manager), the initiating queue manager opens one or more channels and applies the level of protection (MQeAttribute) to the channel that matches the level of protection (MQeAttribute) defined for the target queue. In order to minimize the opening of multiple channels, (for example one per target queue), the initiating queue manager attempts channel reuse according to the selected channel attribute rules.

The rules used depend on the setting of the ″ChannelAttrRules″ keyword in the configuration file used at queue manager creation, (for example in

MQePrivateClient.ini or MQePrivateServer.ini before SimpleCreateQM was used). By default, this is set to use a supplied example:

```
ChannelAttrRules=examples.rules.AttributeRule
```

Before reusing a channel, the initiating queue manager determines whether the current channel attribute is sufficient to protect a message to the given target queue. To do this, it uses the current AttributeRule `equals` method. This method checks if the channel attribute is equal or better than the target queue attribute. If equal or better it reuses the channel, if not, it attempts to dynamically upgrade the channel to the relevant level of protection by upgrading the channel attribute. If this is successful, it reuses the channel, if not successful, it opens and uses a new channel with the required level of protection.

The upgrade mechanism uses the current AttributeRule `permit` method to determine if channel attribute upgrade is permitted. The examples.rules.AttributeRule `permit` method allows upgrade from weaker to stronger or equivalent levels of protection, but not vice versa.

Before allowing channel reuse, the target queue manager uses its current AttributeRule `equals` method to determine if the current channel attribute can provide an appropriate level of protection for the target queue.

While the examples.attributes.AttributeRule provides practical defaults, there may be many solution specific reasons why different behavior is required. The MQSeries Everyplace based solution creator can achieve this by extending or replacing the default examples.attribute.AttributeRule with rules defining the desired behavior.

It is possible, but not recommended, to run without setting ChannelAttrRules.

Message-level security

Message-level security facilitates the protection of message data between an initiating and receiving MQSeries Everyplace application. Message-level security is an application layer service. It requires the initiating MQSeries Everyplace 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 setup an appropriate, 'matching', message-level attribute and pass it to the receiving queue manager so that the attribute is available when getMessage is used to get the message from the target queue.

Like local security, message-level security exploits the application of an attribute on a message (MQeFields object descendent). The initiating application's queue manager handles the application's putMessage with the message dump method, which invokes the (attached) attribute's 'encodeData' method to protect the message data. The receiving application's queue manager handles the application's getMessage with the message's 'restore' method which in turn uses the supplied attribute's 'decodeData' method to recover the original message data.

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 classically 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 wish to delegate trust for the confidentiality of message data to any intermediate, but would prefer to manage and control trust management directly.

MQSeries Everyplace 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

MQSeries Everyplace 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 MQSeries Everyplace symmetric cryptor and compressor choices. Like local security it requires the attribute's key to be preset before it is supplied as a parameters 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: This class is available only in the high security version of MQSeries Everyplace Version 1.0.

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/validation so 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/decryption ensures that the message can only be restored by the intended recipient. As with MQeMAttribute, it allows use of all available MQSeries Everyplace symmetric cryptor and compressor choices. Chosen for size optimization, the certificates used are mini-certificates based on the WTLS certificate proposed by the WAP forum WTLS Specification. The mutual availability of the information necessary to authenticate (validate signatures) is provided through the MQSeries Everyplace default PKI infrastructure.

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 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 MQSeries Everyplace 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 that MQeMAttrribute suits their needs, while solutions for which one to one (authenticable entity to authenticable entity) transfer and non-repudiation of the message originator are important may find MQeMTrustAttribute is the correct choice.

Usage guide

The following code fragments provide examples of how to protect and unprotect a message using MQeMAttribute and MQeMTrustAttribute

MQSeries Everyplace message-level security using MAttribute

```
/* SIMPLE PROTECT FRAGMENT                                    */
MQeMsgObject msgObj = null;
MQeMAttribute msgA  = null;
long confirmId      = MQe.uniqueValue();
try{
   trace(">>> putMessage to target Q using MQeMAttribute"
                     + " with 3DES Cryptor and key=It_is_a_secret");
   MQe3DESCryptor tdes  = new MQe3DESCryptor( );
   msgA                 = new MQeMAttribute( null, tdes, null );
   MQeKey localkey      = new MQeKey( );
   localkey.setLocalKey( "It_is_a_secret");
   msgA.setKey( localkey );
   msgObj               = new MQeMsgObject( );
   msgObj.putAscii("MsgData","0123456789abcdef....");
```

```
    newQM.putMessage( targetQMgrName, targetQName,
                                          msgObj, msgA, confirmId );
    trace(">>> MAttribute protected msg put OK...");
    }
  catch (Exception e)
    {
    trace(">>> on exception try resend exactly once.à");
    msgObj.putBoolean( MQe.Msg_Resend, true );
    newQM..putMessage( targetQMgrName, targetQName,
                                           msgObj, null, confirmId );
    }

  /* SIMPLE UNPROTECT FRAGMENT                                        */
  {
  MQeMsgObject msgObj2 = null;
  MQeMAttribute msgA   = null;
  long confirmId       = MQe.uniqueValue();
  try{
    trace(">>> getMessage from target Q using MQeMAttribute" +
                        " with 3DES Cryptor and key=It_is_a_secret");
    msgA               = new MQeMAttribute( null, null, null );
    MQeKey localkey    = new MQeKey( );
    localkey.setLocalKey( "It_is_a_secret");
    msgA.setKey( localkey );
    msgObj2            = newQM.getMessage( targetQMgrName,
                               targetQName, null, msgA, confirmId );
    trace(">>> unprotected MsgData = "
                                + msgObj2.getAscii(("MsgData") );
    }
  catch (Exception e)
    {                                  /* exception may have left  */
    newQM.undo( targetQMgrName,        /* message locked on queue  */
             targetQName, confirmId ); /* undo just in case        */
    e.printStackTrace( );              /* show exception reason    */
    }
...}
```

MQSeries Everyplace message-level security using MTustAttribute

```
   /* SIMPLE PROTECT FRAGMENT                                        */
  {
  MQeMsgObject msgObj      = null;
  MQeMTrustAttribute msgA  = null;
  long confirmId           = MQe.uniqueValue();
  try {
    trace(">>> putMessage from Bruce1 intended for Bruce8
       to target Q using MQeMTrustAttribute with MARSCryptor ");
    MQeMARSCryptor mars        = new MQeMARSCryptor( );
    msgA                       = new MQeMTrustAttribute(
                                          null, mars, null);
    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 );
    sendreg.setTargetRegistryName("Bruce8");
    msgA.setPrivateRegistry( sendreg );
    MQePublicRegistry pr       = new MQePublicRegistry( );
    pr.activate("MQeNode_PublicRegistry", ".//" );
    msgA.setPublicRegistry( pr );
    msgA.setHomeServer( MyHomeServer + ":8081" );
    msgObj                     = new MQeMsgObject();
    msgObj.putAscii("MsgData","0123456789abcdef....");
    newQM.putMessage( targetQMgrName, targetQName,
                                      msgObj, msgA, 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, msgA, confirmId );
    }
 }

/* SIMPLE UNPROTECT FRAGMENT                                        */
{
MQeMsgObject msgObj2     = null;
MQeMTrustAttribute msgA  = null;
long confirmId           = MQe.uniqueValue();
try {
    trace(">>> getMessage from Bruce1 intended for Bruce8
       from target Q using MQeMTrustAttribute with MARSCryptor ");
    MQeMARSCryptor mars        = new MQeMARSCryptor( );
    msgA                       = new MQeMTrustAttribute(
                                       null, mars, null);
    String EntityName          = "Bruce8";
    String PIN                 = "12345678";
    Object Passwd              = "It_is_a_secret";
    MQePrivateRegistry sendreg = new MQePrivateRegistry( );
    sendreg.activate( EntityName, ".//MQeNode_PrivateRegistry",
                                    PIN, Passwd, null, null );
    msgA.setPrivateRegistry( sendreg );
    MQePublicRegistry pr       = new MQePublicRegistry( );
    pr.activate("MQeNode_PublicRegistry", ".//" );
    msgA.setPublicRegistry( pr );
    msgA.setHomeServer( MyHomeServer + ":8081" );
    msgObj2                    = newQM.getMessage( targetQMgrName,
                                   targetQName, null, msgA, confirmId );
    trace(">>> MTrustAttribute protected msg =
                                     msgObj2.getAscii("MsgData") );
    }
 catch (Exception e)
    {                                 /* exception may have left  */
    newQM.undo( targetQMgrName,       /* message locked on queue  */
              targetQName, confirmId ); /* undo just in case        */
    e.printStackTrace( );             /* show exception reason    */
    }
 }
```

Private registry service

This section describes the private registry service provided by MQSeries Everyplace.

Private registry and the concept of 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.

MQSeries Everyplace abstracts the concept of *target of authentication* from user (person) to *authenticatable entity*. This does not exclude the possibility of authenticatable entities being people, but this would be an application selected mapping.

Internally, MQSeries Everyplace defines all queue managers that can either originate or be the target of mini-certificate dependent services as an authenticatable entities. MQSeries Everyplace also defines queues defined to use mini-certificate based authenticators as authenticable entities. So queue managers that support these services can have one (the queue manager only), or a set (the queue manager and every queue that uses certificate based authenticator) of authenticatable entities.

MQSeries Everyplace provides configurable options to enable queue managers and queues to auto-register as an autenticatable entity. MQSeries Everyplace private registry service (MQePrivateRegistry) provides services that enable an MQSeries Everyplace 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.

Private registry and authenticatable entity credentials

To be useful every authenticable 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. MQSeries Everyplace 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

MQSeries Everyplace 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 MQSeries Everyplace application uses MQePrvateRegistry 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 and these, 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 ublic key (mini-certificate).

Usage scenario

The primary purpose of MQSeries Everyplace's private registry is to provide a private repository for MQSeries Everyplace 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 MQSeries Everyplace security features:

Queue-based security with MQeWTLSCertAuthenticator
Whenever queue-based security is used, where a queue attribute is defined with MQeWTLSCertAuthenticator, (Mini Cerificate based mutual authentication) the authenticatable entities involved are MQSeries Everyplace 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 MQSeries Everyplace 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 MQSeries Everyplace mini-certificate issuance service) to trigger auto-registration to create the entities' credentials and to save them in the entities' own private registries.

Secure feature choices

MQSeries Everyplace Version 1 provides no support for any alternative secure repository for an authenticatable entity's credentials. If queue-based security with MQeWTLSCertAuthenticator or message-level security using MQeMTrustAttribute are used, private registry services must be used.

Selection criteria

The selection criteria for private registry are the same as those for queue-based and message-level security.

Usage guide

Prior to using queue-based security, MQSeries Everyplace owned authenticabale 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 MQSeries Everyplace mini-certificate issuance service.
2. In administration mode, add the name of the queue manager as a valid authenticatable entity, and the entity's one-time-use certificate request PIN.

3. Start the mini-certificate server in server mode.
4. Configure `MQePrivateClient1.ini` and `MQePrivateServer1.ini`as described in ″Using SimpleCreateQM to create ClientQMgr and ServerQMgr instances″ 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 used 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 MQSeries Everyplace mini-certificate issuance service.
2. In administration mode, add the name of the application entity, and allocate the entity a one-time-use certificate request PIN.
3. Start the mini-certificate server in server Mode.
4. Use a program similar to the program 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:8081";
   /* 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( );
   }
```

Public registry service

This section describes the public registry service provided by MQSeries Everyplace.

MQSeries Everyplace provides default services facilitating the sharing of authenticatable entity *public credentials* (mini-certificates) between MQSeries Everyplace nodes. Access to these mini-certificates is a prerequisite for message-level security. MQSeries Everyplace 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. MQSeries Everyplace 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 MQSeries Everyplace 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.

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 MQSeries Everyplace 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 MQSeries Everyplace client node to automatically get, from its MQSeries Everyplace home server, the mini-certificates of other authenticable entities that it needs, 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 MQSeries Everyplace nodes.

The alternative to this intelligent replication may be to have an out-of-band utility to initialize an MQSeries Everyplace 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 MQSeries Everyplace 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 MQSeries Everyplace 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 MQSeries Everyplace node public registry.

Usage guide

```
/* 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:8081";
    /* auto-register Bruce1, Bruce2...Bruce8                            */
    int i                   = 1;
    for ( i = 1; i < 9; i++ )
      {
      EntityName = "Bruce" + (new Integer(i)).toString( );
      MQePrivateRegistry preg = new MQePrivateRegistry( );
      preg.activate( EntityName, ".\\MQeNode_PrivateRegistry" ,
      EntityPIN, KeyRingPassword, CertReqPIN, CAIPAddrPort);
      /* inst'ate and activate PublicReg & save MiniCert from PrivReg  */
      MQePublicRegistry pubreg = new MQePublicRegistry( );
      pubreg.activate(  "MQeNode_PublicRegistry", ".\\" );
      pubreg.putCertificate( EntityName,
                             preg.getCertificate( EntityName ) );
      /* before share of MiniCert                                      */
      pubreg.shareCertificate( EntityName,
```

```
            preg.getCertificate( EntityName ),  "9.20.X.YYY:8081" );
      preg.close();
      pubreg.close();
      }
    }
catch (Exception e)
    {
    e.printStackTrace( );
    }
```

Note: It is not possible to activate a public registry instance more than once, hence the example above demonstrates the recommended practice of accessing a public registry by creating a new instance of MQePublicRegistry, activating the instance, performing the required operations and closing the instance.

Mini-certificate issuance service

MQSeries Everyplace 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. The characteristics of this issuance service are:

- Management of the set of registered authenticatable entities
- Issuance of mini-certificates (the mini-certificate is based on the WAP WTLS mini-certificate)
- Management of the mini-certificate repository

The tools provided 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 MQSeries Everyplace application or MQSeries Everyplace queue manager) can then be configured to provide this certificate request PIN at start-up 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.

The mini-certificates that have been issued by a mini-certificate issuance service are held in the issuance service's own registry. When a mini-certificate is reissued (for example as the result of expiry), the expired mini-certificate is archived.

Configuring, starting and ending an instance of mini-certificate issaunce service server

Configuration using MQSeries EveryplaceMiniCertificateServer.ini

`MQeMiniCertificateServer.ini` is an example configuration file. Instances of MQeMiniCertificateServer can be created by modifying this example, and using it at MQeMiniCertificateServer start-up time. `MQeMiniCertificateServer.ini` includes Alias, ChannelManager,Listener and MiniCertSvrRegistry sections. An instance of MQeMiniCertificateServer uses the contents of these sections at start-up to auto-configure its behavior.

MQeMiniCertificateServer.ini is an extension of ExamplesMQeServer.ini. The extensions are described here, for all other options, please refer to the description of ExamplesMQeServer.ini.

Extension to [Alias] Section
Two mandatory keywords are added:

MiniCertSvrRegistry
This setting identifies the class name of the registry to be used

MiniCertIssuanceManager
This setting identifies the name of the class that implements the MQeMiniCertIssuanceInterface

Additional [MiniCertServerRegistry] Section
This section contains two optional keywords:

PIN This identifies the valid MQeMiniCertificateServer Administrator's PIN, used by the MQeMiniCertificateServer to activate and gain access to its private Registry

KeyRingPassword
This identifies the password or passphrase used to protect private objects stored in the MiniCertificateServer's private registry

Starting MQeMiniCertificateServerGUI

MQeMiniCertificateServerGUI.bat is a simple example start-up file. An instance of MQeMiniCertificateServer can be started by modifying and using this example. The example uses the command:

```
java  com.ibm.mqe.server.MQeMiniCertificateServer <parameter1> <parameter2>
```

where:

<parameter1> = com.ibm.MQe.Server.MCSMessageBundle
(or translated versions of MQeMiniCertificateServer messages ListResourceBundle)
<parameter2> = Examples.Trace.MQeTraceResource
(or translated versions of MQSeries Everyplace base messages ListResourceBundle)

Using the GUI to start the mini-certificate issuance service for the first time

Invocation of MQeMiniCertificateServerGUI.bat results in the following being displayed:

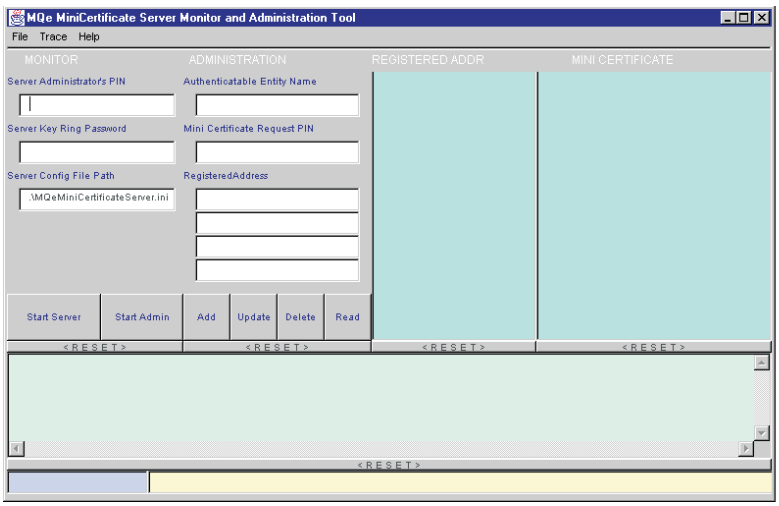

Figure 28. Mini-certificate server GUI

In order to start the mini-certificate server for the first time the administrator needs to:

1. Enter the PIN that is planned for access to this instance of the mini-certificate server in the input 'ServerAdministrator's PIN' field (shown here as '12345678')
2. Enter the password or passphrase that the administrator plans to use to protect the private objects in the mini-certificate server's registry in the ServerKey Ring Password field (shown here as 'It_is_a_secret')
3. Enter the path and filename of the start-up configuration file in the Server Config File Path field (shown here as './MQeMiniCertificateServer.ini')
4. Click the Start Server button

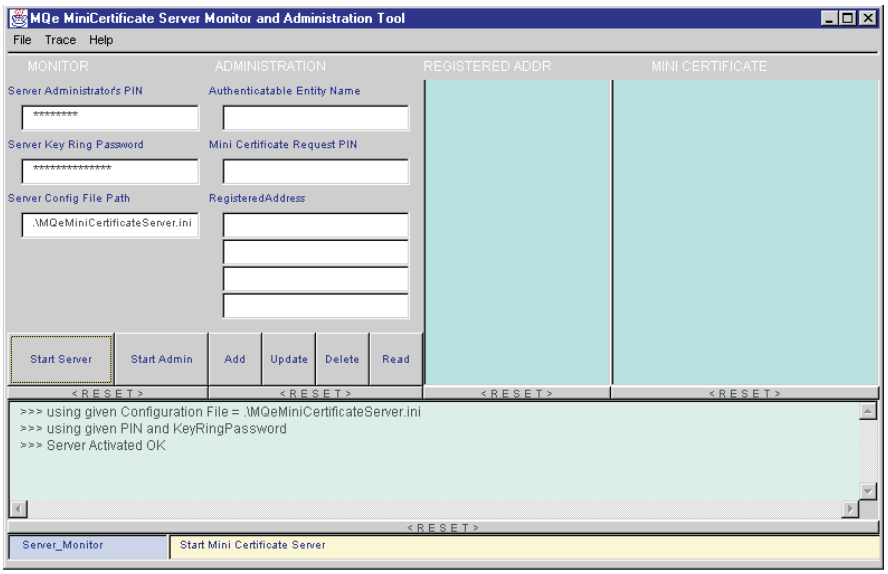

Figure 29. Mini-certificate server started

Note: The Mode indicator in the bottom left of the GUI indicates that the server is started showing 'Server_Monitor'. The Context output to the right of the

mode indicator shows the contextual help for the start server button. The Monitor output above the Mode and Context is an example of valid monitor output.

Using administration tools

Starting administration mode

In order to use the administration tools, the MQeMiniCertificateServerGUI must be invoked and Administration mode started. This can be achieved by invoking `MQeMiniCertificateServerGUI.bat`, filling in the 'Server Administrator's PIN', the 'ServerKey Ring Password' and 'Server Config File Path' input fields, and then selecting the 'Start Admin' button. An example of the visual feedback from this task is:

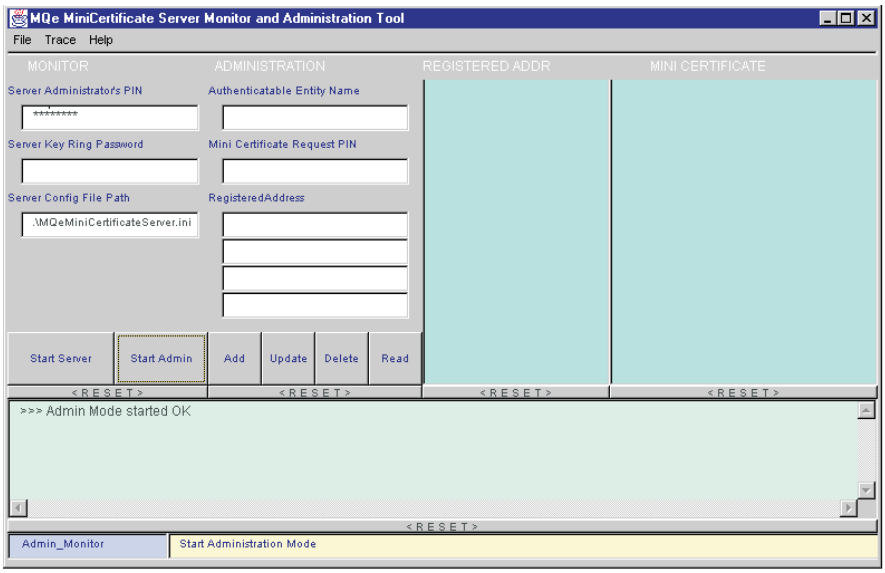

Figure 30. Mini-certificate server administration mode

Adding a new authenticatable entity

Having started administration mode, adding a new authenticatable entity consists of supplying the entity's name and address in the appropriate input fields, and then setting the one-time-use certificate request PIN and clicking the 'Add' button. An example of the visual feedback from this task is:

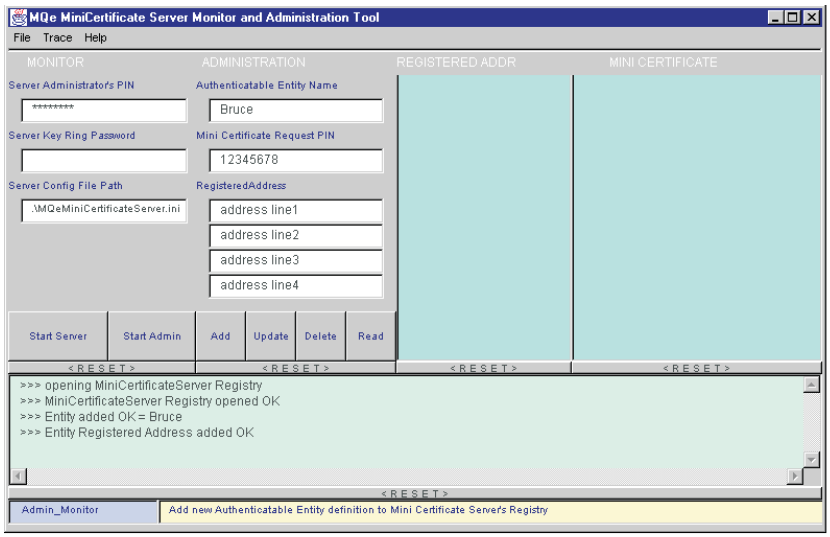

Figure 31. Adding a new authenticatable entitiy

Updating an authenticatable entity

Updating a registered authenticatable entity's details is similar to adding an entity. Having entered administration mode, the authenticatable entity's updated details are provided This can include a new certificate request PIN, if apropriate. Then to update click the Update button. An example of the visual feedback from this task is:

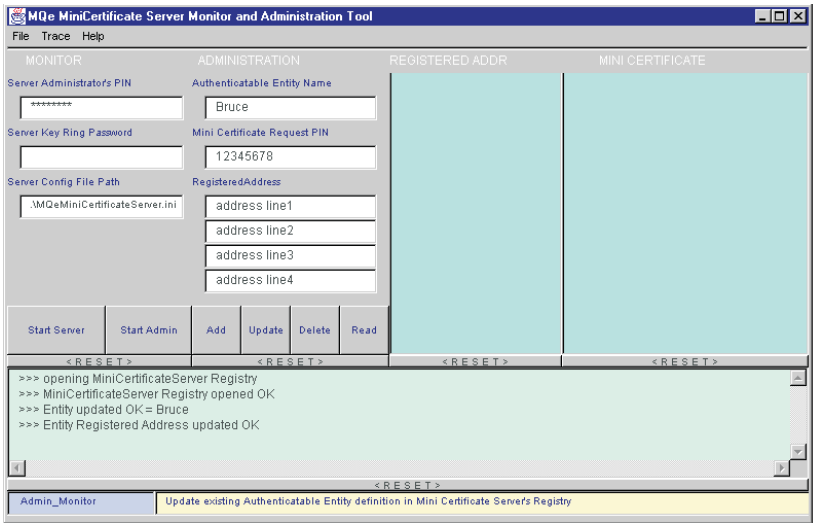

Figure 32. Updating an authenticatable entitiy

Deleting an authenticatable entity

Deleting a registered authenticatable entity's details is achieved by entering the authenticatable entity's name in the input field and then clicking the 'Delete' button.

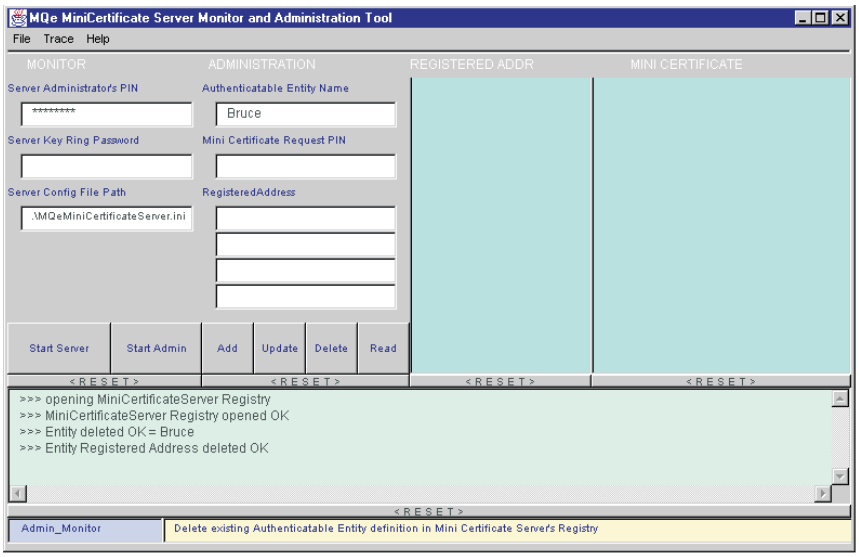

Figure 33. Deleting an authenticatable entitiy

Reading an authenticatable entity's details

To read a registered authenticatable entity's details, enter the authenticatable entity's name in the input field and then click the 'Read' button. An example of the visual feedback from this task is:

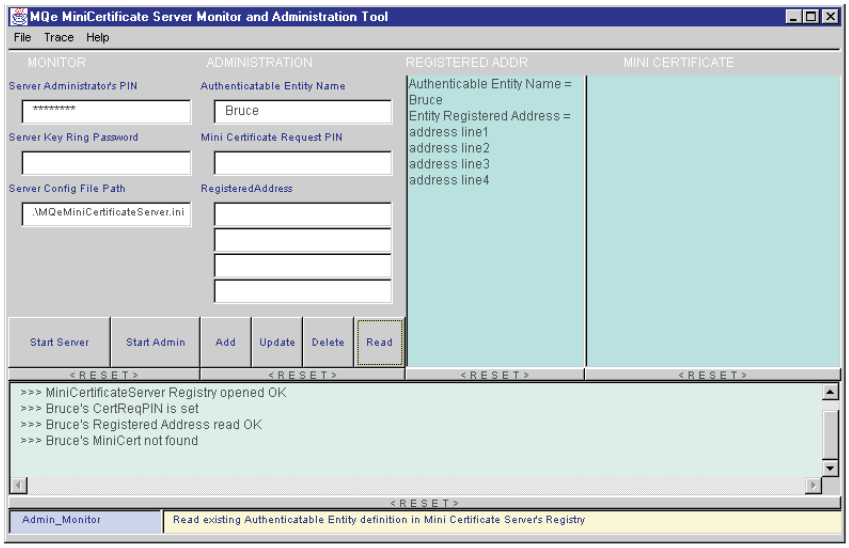

Figure 34. Reading an authenticatable entitiy

This provides a method for displaying the details of any registered authenticatable entity. The visual feedback displays the registered address and mini-certificate, if available and the status of the one-time-use certificate request PIN. In normal use, after an authenticatable entity is registered but before a mini-certificate has been issued, the registered address is displayed, the status of the certificate request PIN is set, and the mini-certificate status is not found. After a mini-certificate has been issued, the registered address and current mini-certificate are displayed and the request PIN status is not set.

Use of File menu Open option

In addition to Read, the Open option is provided to select an authenticatable entity that does not require a name to be entered. To use this option, in administration mode:

1. From the 'File' pull down menu, select the 'Open' option

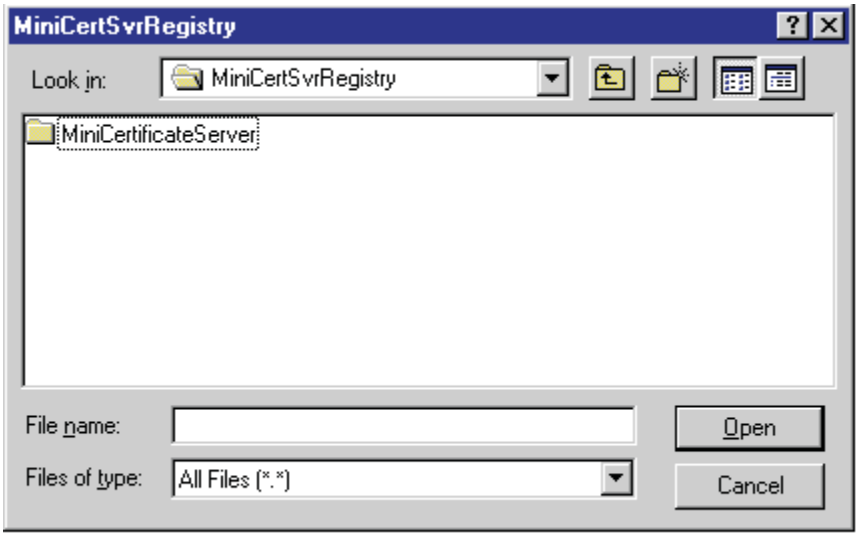

Figure 35. MQSeries Everyplace authenticatable entity details display

2. Select the 'EntityAddr' folder from the displayed list and click on the 'Open' button

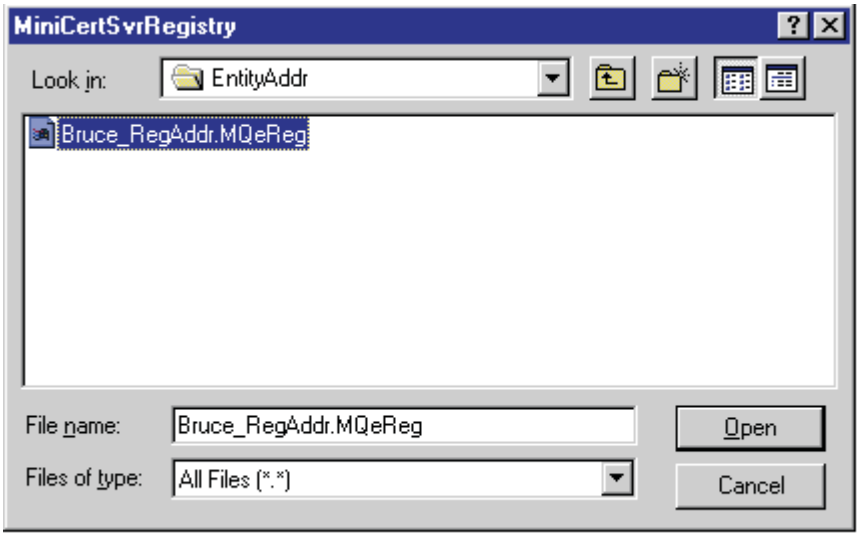

Figure 36. MQSeries Everyplace authenticatable entity details display

3. Select the name of the entity that you want to query from the displayed list and click on the 'Open' button

The entity details are displayed as shown in Figure 37 on page 194.

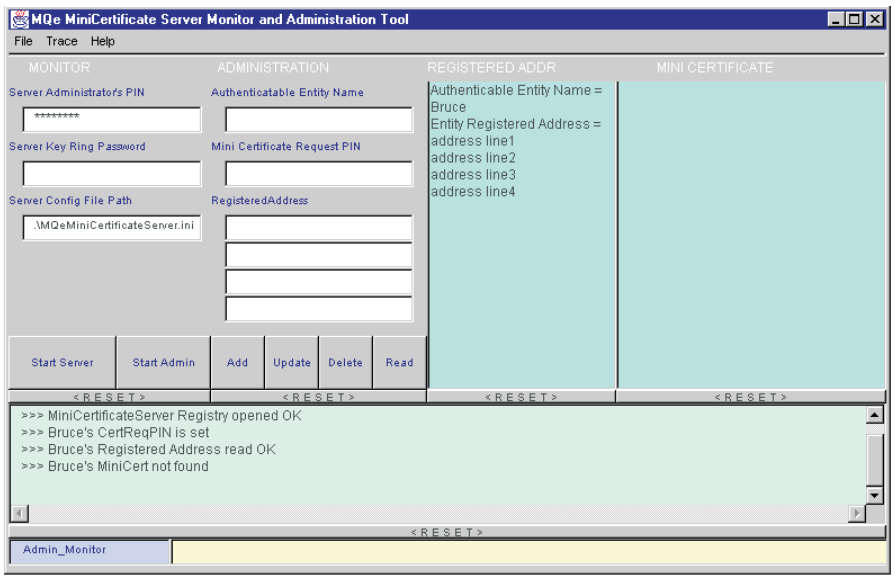

Figure 37. MQSeries Everyplace authenticatable entity details display

Operation

Starting and stopping

Starting an instance of MQeMiniCertificateServerGUI, and using the GUI to either start the server, or to start Administration Mode is described in “Starting MQeMiniCertificateServerGUI” on page 188 and “Starting administration mode” on page 190. In both cases, to terminate the MQeMiniCertificateServerGUI instance, on the ’File’ pull down menu select the ’Exit’ option. select ’yes’ in the confirmation dialog to complete the shutdown of the min-certificate server.

Monitor and logging

When running the server in Server_Monitor mode or in Admin_Monitor mode, the significant events are monitored and visual feedback is provided in the Monitor listbox, with the ’>>> ’ prefix.

An additional option is available in both Server_Monitor mode and Admin_Monitor mode to log these events to a designated file. Operational solutions are likely to use this option to provide an audit trail. To start this option, in either mode, select the ’Log’ option from the File pull down menu. This task results in a file selection dialog box being displayed:

To select a log file name (in which subsequent monitor events are recorded) the administrator must either accept the MQSeries Everyplace generated log file name that appears in the ’File Name’ input field, (in this example ’949679065895_MCSlog’) or overwrite it with a preferred Log filename, then click the ’Save’ button.

An example of the visual feedback from this task is shown in Figure 38 on page 195.

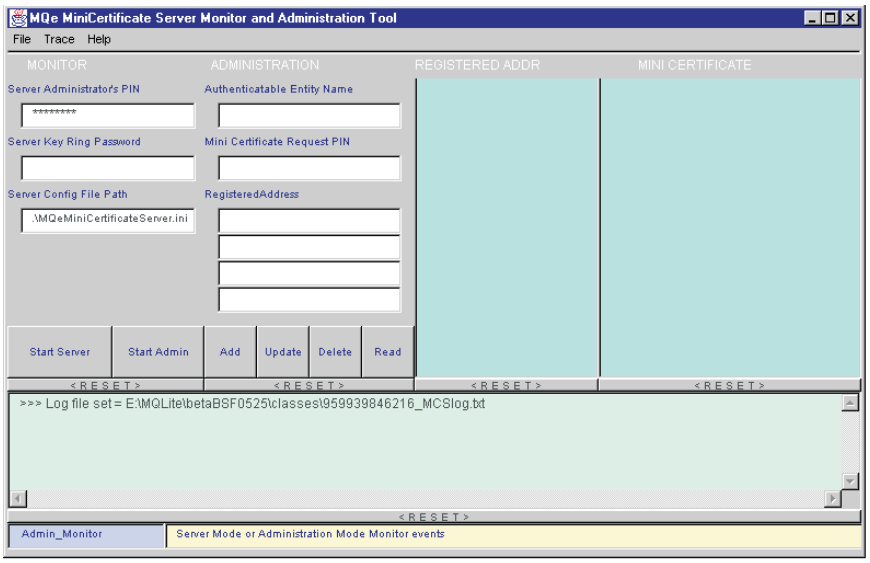

Figure 38. Mini-certificate server log file name display

An example of the Log file created using administration to add an authenticatable entity named 'Bruce' is:

```
>>> Log file set = E:\MQLite\betaBSF0202\Classes\949682538438_MCSlog.txt
>>> Admin Mode started OK
>>> opening MiniCertificateServer Registry
>>> MiniCertificateServer Registry opened OK
>>> Entity added OK = Bruce
>>> Entity Registered Address added OK
```

mini-certificate issuance service

Chapter 9. Tracing in MQSeries Everyplace

This section provides assistance with using and customizing the MQSeries Everyplace trace program.

MQSeries Everyplace provides a simple, but useful, tracing facility. This facility can be used to follow the course of execution of a program either when it is running, or later by inspecting the trail of execution recorded in a file. Trace messages are sent from the running code to a trace window, where they are displayed.

The trace facility is just a trace, it does not contain some features found in debuggers, such as the ability to set and release break points.

The following MQSeries Everyplace example trace classes can be found in the `examples.trace` subdirectory:

MQeTrace
This class is a simple trace handler that displays trace messages on the Java console.

MQeTraceResource
This class contains the templates for all the MQSeries messages

MQeTraceResourceGUI
This class contains all the translatable text for the trace window controls

The following MQSeries Everyplace trace classes can be found in the `examples.awt` subdirectory. These classes can be used to create a graphical user interface to view the trace output.

AwtDialog
This class creates and handles simple dialog style windows

AwtFormat
This class creates and manages the various graphical user interface components within a dialog window or within an Application Windowing Toolkit frame

AwtFrame
This class creates and handles a very simple frame style window

These classes can be used to handle and display trace from a running MQSeries Everyplace environment. Tracing would not normally be used in a production environment, except for diagnosis of problems, as any form of tracing affects the performance of MQSeries Everyplace.

Using trace

To trace the execution of an application program you must put a statement in an appropriate place in the code using the `MQe.trace` method as shown in the following example:

```
...
/* */
trace( "We got here" );
...
```

© Copyright IBM Corp. 2000

When executed, this results in the text ″We got here″ being displayed in the MQSeries Everyplace Trace window.

Trace message formats

There are several types of message (information, warning, error, security and debug) and the type is denoted by the first characters as shown in Table 12.

Table 12. Trace message types

Initial character	Meaning
I or i	Information
W or w	Warning
E or e	Error
S or s	Security
D or d	Debug

Upper case prefixes are used for application trace messages and lower case prefixes are used for system trace messages. System trace messages are usually only generated from within MQSeries Everyplace.

The message is sent to the MQSeries Everyplace trace facility, which checks the level of the message and, if required, outputs it to the trace window. Trace messages that have a recognizable prefix are written to System.err, others are written to System.out

The `examples.trace.MQeTrace` file in the `examples.trace` directory contains the various message templates for the messages issued by MQSeries Everyplace internal routines. The messages are of the form:

```
/* common messages */
    {    "1",            "d;[00001]:Created"                    },
    {    "2",            "d;[00002]:Destroyed"                  },
    {    "3",            "d:[00003]:Close"                      },
    {    "4",            "w:[00004]:Warning:#0"                 },
    {    "5",            "e:[00005]:Error:#0"                   },
    {    "6",            "i:[00006]:Command:#0"                 },
    {    "7",            "i:[00007]:Waiting"                    },
    {    "8",            "i:[00008]:#0 input byte count=#1"     },
    ...,
```

where the first character string is the message number and the second string is the message template.

`examples.trace.MQeTraceResource` contains the message strings in English. Various other language versions are also provided in this directory.

The template has the following format:

- The message type as described in Table 12
- A modifier character, this modifier has the following meanings:

Table 13. Trace message modifiers

Modifier	Meaning
:	no modification applied
;	RESERVED for create/destroy object
+	log this message via the Log interface
¬	ignore - Do not display this message

- The message number in the format '[nnnnn]:'
- The message text. This can include inserts of the form '#n' where 'n' is an integer from 0 to 9

By modifying this source file you can change the classification of a message. For example, you can change from a Warning to an Error, or by changing the modifier character from ':' to '+', you can cause the message to be copied to the Event log.

New trace messages can be added at runtime using the addMessage or addMessageBundle calls. For example, to add a single new message :

```
...
MQeTraceInterface MyTrace = MQe.GetTraceHandler();
myTrace.addMessage(" :[11111]:My Application - #0 = #1" );
...
trace( 11111, new String[] { "Magic word", "xyzzy" } );
...
```

Activating trace

Trace, which is not active by default, can be activated using the `MQe.setTraceHandler` as shown in the following code:

```
...
  /* give the trace object to MQe */
  setTraceHandler( new myTraceHandler() );
  trace( "I:Starting..." );
  ...
```

The example trace handler that is shipped as part of the MQSeries Everyplace toolkit, includes the trace activation code.

Customizing trace

The trace classes provided in the examples directory can be used as a basis for custom trace handlers.

MQeTrace example

The MQeTrace example class provides a simple, tracing facility that by default outputs the trace messages to System.out and/or to System.err.

To activate the trace window specify the following code:

```
...
/* Start the example version of MQeTrace  */
new examples.trace.MQeTrace( "Trace", null ) );
...
trace( "I:Starting..." );
trace( 123456, "Insert" );
...
```

The second parameter on the constructor is the language to be used for the trace messages, if null is specified, the default language is used. Alternatively a different resource file may be specified that changes the classification of the messages, for example:

```
...
/* Start the example version of MQeTrace  */
new examples.trace.MQeTrace( "Trace", "MyMessageResourceFile") );
```

```
...
trace( "I:Starting..." );
trace( 123456, "Insert" );
...
```

The currently active trace handler object can be found by issuing an `MQe.getTraceHandler` method call. Using this reference the behavior of the trace can be modified, that is selecting or deselecting the types of trace messages to be written.

```
...
  /* Start the example version of MQeTrace  */
  MQeTraceInterface trace = MQe.getTraceHandler( );
  if ( trace instanceof MQeTrace )
    {
    ((MQeTrace) trace).MsgInf   = true;
    ((MQeTrace) trace).MsgDebug = true;
    ((MQeTrace) trace).MsgTime  = true;
    }
  ...
  trace( "I:Starting..." );
  ...
```

The variables (and their defaults) in MQeTrace that may be modified are :

```
public   boolean   MsgInf       = false;      /* Informaton msgs      */
public   boolean   MsgWarn      = true;       /* warning msgs         */
public   boolean   MsgErr       = true;       /* error msgs           */
public   boolean   MsgSecurity  = false;      /* Security  msgs       */
public   boolean   MsgSys       = true;       /* System modifier      */
public   boolean   MsgDebug     = false;      /* Debug modifier       */
public   boolean   MsgLog       = false;      /* Trace message to log */
public   boolean   MsgTime      = false;      /* add Time stamp       */
public   boolean   MsgPrefix    = false;      /* add object prefix    */
public   boolean   MsgThread    = false;      /* add Thread ID        */
```

More details can be found by examining the source code for MQeTrace in the `examples.trace` directory:

This trace example can be used as the basis for a more sophisticated trace program or a completely new one could be created.

The application program could even be the trace handler as well as its normal function just by implementing the MQeTraceInterface and issuing the `MQe.setTraceHandler` method call.

Graphical user interface for trace

The basic trace function provided in the `examples.trace` directory just displays the trace messages on `System.out` and `System.err` in the console window associated with the application.

There is another trace handler supplied in the `examples.awt` directory that uses a subset of the Java AWT to provide a graphical user interface to the Trace, this enables the various tracing options to be modified dynamically.

```
...
/* Start the example GUI version of MQeTrace */
new examples.awt.AwtMQeTrace( "My Trace title", null ) );
...
trace( "I:Starting..." );
```

This code starts the trace window with the title 'My Trace' and displays the information message "I:Starting". The trace window has pull-down menus that

enable the user to modify the level of tracing, the format of the messages, and other properties, as shown in Figure 39. Note that an MQSeries Everyplace object is required to perform tracing. The examples above assumed that the code is part of a class that extends the base MQSeries Everyplace class. It is possible to output MQSeries Everyplace trace messages from objects that do not themselves extend MQSeries Everyplace. In this case, you need to create an MQSeries Everyplace object, and then specify the tracing by using the methods of this object. For example:

```
...
/* create a MQe object */
MQe dbg = new MQe( );
dbg.Message( "D:We got here" );
...
```

MQSeries Everyplace tracing is Java virtual machine wide, so that all messages from MQSeries Everyplace objects executed on any thread in the current Java Virtual Machine are handled by the same trace facility, and displayed in the same trace window. This can be a big advantage as it shows the order in which events actually occurred. However it can be a disadvantage if you wish to separate out totally independent events occurring on different threads.

Note: Terminating the MQSeries Everyplace Trace window does not terminate the Java program.

Example AWT trace window layout

Note that you need a `MyMessageResourceFileGUI` file that specifies the text to be used in any graphical user interface components associated with trace.

The example trace program in `examples.awt` produces a window with the layout shown in Figure 39.

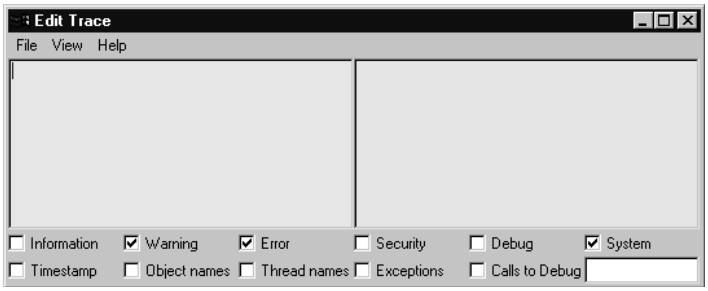

Figure 39. Example trace GUI window

The Menu items are:

- File Menu:

  Clear clears the trace window

  Save As...
  Save the contents of the trace window to a disk file

  Trace to Log
  Copy Trace messages to the event log

Trap I/O
Output to System.out and System.err is displayed in the window. If this option is unchecked output goes to the Java console

Kill Terminate both trace and the owning application. Clicking on the window frame exit button only terminates the Trace

- View Menu:

View Options
Show the trace message display options

System.out
Show the System.out window

System.err
Show the System.err window

The various trace message display options control how, and which, trace messages are to be displayed in the *System.err.println* window:

Information
Display information messages

Warning
Display warning messages

Error Display error messages

Security
Display security messages

Debug
Display debugging information messages

System
Controls whether messages with the System characteristics are displayed. This affects Information, Warning, Error, Security and Debug style messages

Timestamp
Prefixes the messages with the current time stamp

Object names
Prefixes the messages with the object type and instance that originated the message

Thread names
Prefixes the messages with the name of the thread that is running at the time

Exceptions
Causes a stack trace to be displayed whenever an MQeException is thrown

Calls to Debug
Causes a stack trace to be displayed whenever an application or MQSeries Everyplace issues the MQe.Debug call

The 'System.err.println and Trace message filter' is a string that is used to match anything within the output. If the match is successful the output is displayed, if the match was unsuccessful the output is not displayed.

Using this feature it is possible to selectively display messages from a specific thread (assuming that the Thread name checkbox is checked).

Setting trace options

The various trace options can be preset on start-up of the AwtMQeTrace program by creating a new GUI resource file that prechecks any of the checkable components for example:

```
public class MQeTraceResourceGUI extends java.util.ListResourceBundle
  {
  static final Object[][] contents = {
  /*    Check items can be pre-checked by replacing the blank with an "!"  */
          { "File",            "File"           },
          {   "Clear",         "Clear"          },
          {   "Save",          "Save As..."     },
          {   "Log",           " Trace to Log"  },    /* check item      */
          {   "Trap",          "!Trap I/O"      },    /* check item      */
          {   "Halt",          "Kill"           },
          { "View",            "View"           },
          {   "Options",       "!View Options"  },    /* check item      */
          {   "SystemOut",     "!System.out"    },    /* check item      */
          {   "SystemErr",     " System.err"    },    /* check item      */
          { "Help",            "Help"           },
          {   "About",         "About..."       },
  /* checkbox labels                                                      */
          { "Information",     " Information"   },    /* check item      */
          { "Warning",         "!Warning"       },    /* check item      */
          { "Error",           "!Error"         },    /* check item      */
          { "Debug",           " Debug"         },    /* check item      */
          { "Security",        " Security"      },    /* check item      */
          { "System",          "!System"        },    /* check item      */

          { "Timestamp",       " Timestamp"     },    /* check item      */
          { "Objects",         " Object names"  },    /* check item      */
          { "Threads",         " Thread names"  },    /* check item      */
          { "Exceptions",      " Exceptions"    },    /* check item      */
          { "CallStack",       " Calls to Debug" },   /* check item      */

  /* About dialog                                                         */
   { "AboutTitle",      "About MQe Trace" },
   { "AboutVersion",    "MQe version"     },
   { "AboutProduct",    "Product number 5639-I47"     },
   { "AboutCopyright",  "(C) Copyright IBM Corp. 1999 All Rights Reserved" },
   { "AboutCopyright2", "Licensed Materials - Property of IBM" },
   { "AboutTrace",      "Trace version"   },
   { "AboutComments",   " "               },
   { "OK",              "OK"              },

                                    };

  public Object[][] getContents( )
    {
    return( contents );
    }
```

Note: If trace options are modified programmatically for MQeTrace, as shown in the following code, the corresponding components on the AwtMQeTrace window WILL NOT be updated

```
...
/* Start the example version of MQeTrace  */
MQeTraceInterface trace = MQe.getTraceHandler( );
if ( trace instanceof MQeTrace )
  ((MQeTrace) trace).MsgDebug = true;
...
```

Chapter 10. MQSeries Everyplace adapters

This section provides information on the creation of MQSeries Everyplace adapters. The coding of two adapters is described, one for communications and one for storing a message.

Please refer to Chapter 9 in the MQSeries Everyplace for Multiplatforms, Programming Reference for details of the communications adapter classes that are supplied with MQSeries Everyplace.

An example of a simple 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 MQSeries Everyplace 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 MQSeries Everyplace 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.

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
    {
```

© Copyright IBM Corp. 2000

```
    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 MQSeries Everyplace 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 MQSeries Everyplace 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 examples 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. Care needs to be taken to 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 of a simple 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 MQSeries Everyplace but rather as a simple introduction into how to create 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;           /* open'd for reading  */
  protected boolean writing  = false;
```

Because this adapter implements FilenameFilter the following method must be coded. This is the flittering 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.

Appendix A. Applying maintenance to MQSeries Everyplace

To apply a maintenance update follow the instructions provided with the update.

For more general information on maintenance updates and their availability see the MQSeries family Web page at `http://www.software.ibm.com/ts/mqseries/`.

© Copyright IBM Corp. 2000

Appendix B. 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,
Hursley Park,

© Copyright IBM Corp. 2000

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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of International Business machines Corporation in the United States, or other countries, or both.

AIX
IBM
MQSeries

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.

UNIX is a registered trademark of X/Open in the United States and other countries.

Windows and Windows NT are registered trademark of Microsoft Corporation in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

This glossary describes terms used in this book and words used with other than their everyday meaning. In some cases, a definition may 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, see the index or the *IBM Dictionary of Computing*, New York:. McGraw-Hill, 1994.

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 communicating 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 checks that verifies the senders and receivers of messages.

bridge. An MQSeries Everyplace object that allows messages to flow between MQSeries Everyplace and other messaging systems, including MQSeries.

channel. See *dynamic channel* and *MQI channell*.

channel manager. An MQSeries Everyplace object that supports logical multiple concurrent communication pipes between end points.

class. A class is 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 MQSeries, a client is a run-time component that provides access to queuing services on a server for local user applications.

compressor. A program that compacts a message to reduce the volume of data to be transmitted.

cryptor. A program that encrypts a message to provide security during transmission.

dynamic channel. A dynamic channel connects MQSeries Everyplace devices and transfers synchronous and asynchronous messages and responses in a bidirectional manner.

encapsulation. Encapsulation is 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.

gateway. An MQSeries Everyplace gateway (or server) is a computer running the MQSeries Everyplace code including a channel manager.

Hypertext Markup Language (HTML). A language used to define information that is to be displayed on the World Wide Web.

instance. An instance is an object. When a class is instantiated to produce an object, we say that the object is an instance of the class.

interface. An interface is 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. The Internet is 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).

Java Developers 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.

Lightweight Directory Access Protocol (LDAP). LDAP is a client-server protocol for accessing a directory service.

Local area network (LAN). A computer network located on a user's premises within a limited geographical area.

message. In message queuing applications, a message is a communication sent between programs.

message queue. See queue

© Copyright IBM Corp. 2000

message queuing. A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

method. Method is the object-oriented programming term for a function or procedure.

MQI channel. An MQI channel connects an MQSeries client to a queue manager on a server system and transfers MQI calls and responses in a bidirectional manner.

MQSeries. MQSeries is a family of IBM licensed programs that provide message queuing services.

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 MQSeries, an object is a queue manager, a queue, or a channel.

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 addistant (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

queue. A queue is an MQSeries object. Message queueing applications can put messages on, and get messages from, a queue

queue manager. A queue manager is a system program the provides message queuing services to applications.

server. (1) An MQSeries Everyplace server is a device that has an MQSeries Everyplace channel manager configured. (2) An MQSeries 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. (3) The computer on which a server program runs.

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*.

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.

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.

Bibliography

Related publications:

- *MQSeries Everyplace for Multiplatforms, Introduction*, GC34-5843-00
- *MQSeries Everyplace for Multiplatforms, Programming Reference*, SC34-5846-00
- *MQSeries An Introduction to Messaging and Queuing*, GC33-0805-01
- *MQSeries for Windows NT V5R1 Quick Beginnings*, GC34-5389-00

© Copyright IBM Corp. 2000

Index

© Copyright IBM Corp. 2000

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–870229
  - From within the U.K., use 01962–870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

© Copyright IBM Corp. 2000

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5845-01