WebSphere MQ Everyplace

C Bindings Programming Guide

Version 2.0.0.5

SC34-6280-01

WebSphere MQ Everyplace

C Bindings Programming Guide

Version 2.0.0.5

SC34-6280-01

Take Note!

Before using this information and the product it supports, be sure to read the general information under Appendix B, "Notices", on page 137

Second Edition (April 2003)

This edition applies to WebSphere® MQ Everyplace™ Version 2.0.0.5 (Program number: 5724-C77) 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 WebSphere MQ family library Web page at `http://www.ibm.com/software/ts/mqseries/library/`.

Contents

About this book

This book is a programming guide for the WebSphere MQ Everyplace product. It contains information on how to use the WebSphere MQ Everyplace C class libraries, that are described in , and the client platform C APIs that are described in the WebSphere MQ Everyplace C Programming Reference.

It provides guidance to help you to decide which classes or APIs to use for common messaging tasks, and in many cases example code is supplied.

For more information on writing C, see the *WebSphere MQ Everyplace C Programming Reference*, on the product CD. The C APIs discussed in this manual are intended to provide client platform functionality. For information on writing C-based programs that provide server platform functionality, refer to the

Chapter 1, "Overview", on page 1 provides a brief introduction for those who are unfamiliar with the concepts and components of WebSphere MQ Everyplace.

To program in C, uset this book in conjunction with the WebSphere MQ Everyplace C Programming Reference.

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

License warning

WebSphere MQ Everyplace Version 2.0.0.5 is a toolkit that enables users to write WebSphere MQ Everyplace applications and to create an environment in which to run them.

Before deploying this product, or applications that use it, in a production environment, please make sure that you have the necessary licenses.

To use WebSphere MQ Everyplace on specified server platforms, other than for purposes of code development and test, you must obtain capacity-unit Use Authorizations so that you are licensed to use the program on each machine and machine upgrade. These licences are recorded on Proof of Entitlement documents and authorize the use of WebSphere MQ Everyplace according to published capacity unit and pricing group tables.

You require device platform use authorizations to use the product (other than for purposes of code development and test) on specified client platforms. These licences are recorded on Proof of Entitlement documents and authorize the use of WebSphere MQ Everyplace. However, they do not entitle you to use the WebSphere MQ Everyplace Bridge, or to run on the server platforms specified in the WebSphere MQ Everyplace pricing group lists published by IBM® and also available on the Web via the following URL:

Please refer to *http://www.ibm.com/software/mqseries* for details of these restrictions.

Who should read this book

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

For information on the availability of development kits for programming languages other than C, see the WebSphere MQ Web site at http://www.ibm.com/software/mqseries/

Prerequisite knowledge

This book assumes that the reader has a working knowledge of C programming.

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 WebSphere MQ books:

- *WebSphere MQ An Introduction to Messaging and Queuing*
- *WebSphere MQ for Windows NT® V5R1 Quick Beginnings*, or the WebSphere MQ Quick Beginnings book that is relevant to the operating system that you are using.

These books are available in softcopy form from the Book section of the online WebSphere MQ library. The library can be reached from the WebSphere MQ Web site, URL address *http://www.ibm.com/software/mqseries/library/*

Terms used in this book

The following terms are used throughout this book:

WebSphere MQ family
refers to the following WebSphere MQ products:

- **WebSphere MQ Workflow** simplifies integration across the whole enterprise by automating business processes involving people and applications
- **WebSphere MQ Integrator** is powerful message-brokering software that provides real-time, intelligent rules-based message routing, and content transformation and formatting
- **WebSphere MQ Messaging** provides any-to-any connectivity from desktop to mainframe, through business quality messaging, with over 35 platforms supported

WebSphere MQ Messaging
refers to the following messaging product groups:

- **Distributed messaging:** WebSphere MQ for Windows NT, AIX®, AS/400®, HP-UX, Sun Solaris, and other platforms
- **Host messaging:** WebSphere MQ for OS/390®
- **Workstation messaging:** WebSphere MQ for Windows
- **Pervasive messaging:** WebSphere MQ Everyplace

WebSphere MQ
refers to the following three WebSphere MQ Messaging product groups:

- Distributed messaging
- Host messaging

- Workstation messaging

WebSphere MQ Everyplace
Refers to the fourth WebSphere MQ Messaging product group, pervasive messaging.

Device platform
A small computer that is capable of running WebSphere MQ Everyplace only as a client.

Server platform
A computer of any size that is capable of running WebSphere MQ Everyplace as a server or client.

Gateway
A computer of any size running WebSphere MQ Everyplace programs that include the WebSphere MQ bridge function.

Chapter 1. Overview

WebSphere MQ Everyplace code can run on a large range of platforms including pervasive and mobile devices. Unlike base WebSphere MQ, WebSphere MQ Everyplace has a single queue manager type. However, WebSphere MQ Everyplace queue managers can be programmed to act as traditional clients or servers.

The fundamental elements of the WebSphere MQ Everyplace programming model are *messages*, *queues* and *queue managers*. WebSphere MQ Everyplace messages are objects that contain application-defined content. When stored, they are held in a queue and such messages may be moved across an WebSphere MQ Everyplace network. Queues can either be local or remote and are managed by queue managers.

WebSphere MQ Everyplace queue managers communicate through WebSphere MQ Everyplace connections. These connections are created on demand and are referred to as *dynamic*, differentiating them from WebSphere MQ connections which have to be explicitly created. They can also be configured in two different ways, in *peer-to-peer* mode, and in *client/server* mode (see "WebSphere MQ Everyplace connections" on page 5).

The WebSphere MQ bridge component also supports WebSphere MQ client channels to enable WebSphere MQ Everyplace networks to communicate with WebSphere MQ networks.

Figure 1 shows an example of an WebSphere MQ Everyplace network linked to a WebSphere MQ server and the following sections of this chapter give brief descriptions of WebSphere MQ Everyplace objects and their uses.

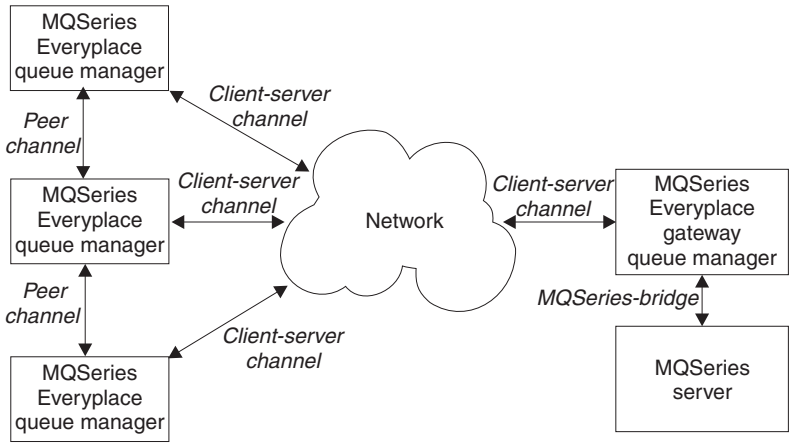

Figure 1. WebSphere MQ Everyplace client

WebSphere MQ Everyplace queue manager

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

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

The WebSphere MQ Everyplace queue manager is code imbedded within user written programs and these programs can run on any WebSphere MQ Everyplace supported device or platform.

Queue managers can be configured in a number of different 'styles', the main ones being *client* (also known as peer, or device), server, and gateway. See "Starting queue managers" on page 33 for descriptions of these styles.

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

For more detailed information about WebSphere MQ Everyplace queue managers see Chapter 5, "Queue managers, messages, and queues", on page 27

WebSphere MQ Everyplace queues

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

- Local
- Remote
- Store-and-forward
- Home-server
- WebSphere MQ 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 and secure way (excluding hardware failures or loss of the device). These queues can be used on a standalone queue manager, or on a queue manager that is 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 66.

Remote queue

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

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. WebSphere MQ 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 68.

Store-and-forward queue

A store-and-forward queue stores messages on behalf of other queue managers until they are ready to receive them. This type of queue is normally defined on a server and can be configured to perform either of the following:

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

Store-and-forward queues can hold messages for many target queue managers, or there may be one store-and-forward queue for each target queue manager. For more detailed information about store-and-forward queues, see "Store-and-forward queue" on page 71.

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 73.

WebSphere MQ bridge queue

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

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

For more detailed information about the WebSphere MQ bridge queue, see "WebSphere MQ bridge queue" on page 75.

Dead-letter queue

WebSphere MQ Everyplace has a similar dead-letter queue concept to WebSphere MQ. Dead-letter queues store message that cannot be delivered. However, there are important differences in the manner they are used.

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

  The use of dead-letter queues with an WebSphere MQ bridge needs special consideration, see "Handling undeliverable messages" on page 105 for more details.

Administration queue

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

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

WebSphere MQ Everyplace connections

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

WebSphere MQ Everyplace clients and servers can communicate over two types of connections, *peer channels* and *client/server channels*.

Client/server channels have the following attributes:

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

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

Peer channels have the following attributes:

- They are dymanic (like client/server channels).
- You can establish the connection from either the client-side or the server-side.
- A queue manager can connect to peer channel listeners on many other queue managers, using a separate connection for each peer channel listener.
- Only one other external client or server can establish a peer channel to the queue manager at any one time. This restriction means that normally, you only use this type of connection between clients, as server queue managers usually want to handle multiple incoming requests concurrently.
- They are not generally used over a firewall, as it is difficult to configure peer channels in this environment.
- They are bidirectional and support the full range of functions provided by WebSphere MQ Everyplace, including both synchronous and asynchronous messaging.

  Note: Bidirectional means that the queue managers on each end of the connection can request and pass data over the connection.

Connections can have various attributes or characteristics, such as authentication, cryptography, compression, or the transmission protocol to use. Different

connections can use different characteristics. Each connection can have its own value set for each of the following attributes:

Authenticator
This attribute causes authentication to be performed. This is a security function that challenges the putting application environment or user to prove their identity.

Cryptor
This attribute causes encryption and decryption to be performed on messages passing through the connection. This is a security function that encodes the messages during transit so that you cannot read them without the decoding information.

Compressor
This attribute causes compression and decompression to be performed on messages passing through the connection. This attempts to reduce the size of messages while they are being transmitted and stored.

Destination
The server and port number for the connection.

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

For more detailed information about connections see "Connections" on page 64. Also, for more information about authenticators, compressors, and cryptors, see Chapter 8, "Security", on page 111.

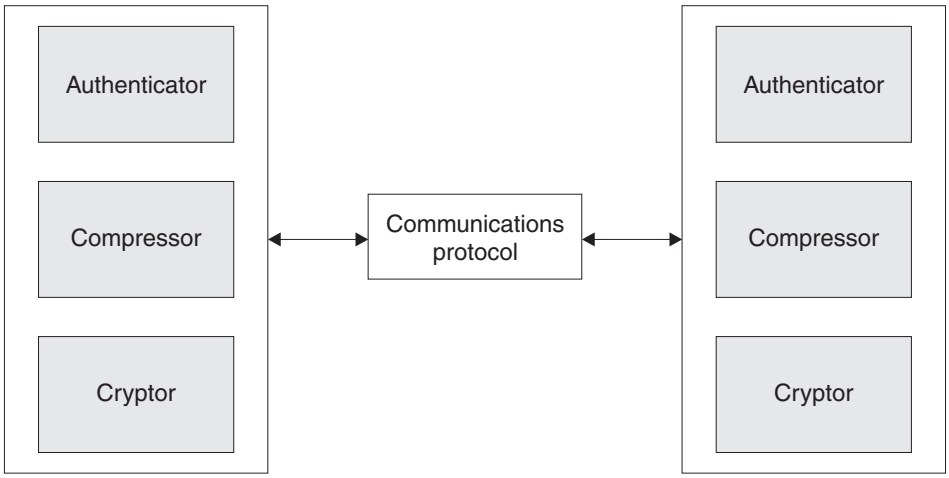

Figure 2. WebSphere MQ Everyplace connection

You can establish WebSphere MQ Everyplace connections using a variety of protocols allowing them to connect in a number of different ways, for example:
- Permanent connection, for example a LAN, or leased line
- Dial out connection, for example using a standard modem to connect to an Internet service provider (ISP)
- Dial out and answer connection, using a CellPhone, or ScreenPhone for example

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

WebSphere MQ Everyplace bridge to WebSphere MQ

An WebSphere MQ Everyplace queue manager can be an interface to a WebSphere MQ server. This type of queue manager is referred to as a gateway queue manager. The WebSphere MQ bridge handles the transfer of messages between the two systems, including the translation between the different message formats. "Configuring the WebSphere MQ bridge" on page 79 provides a detailed description of this interface.

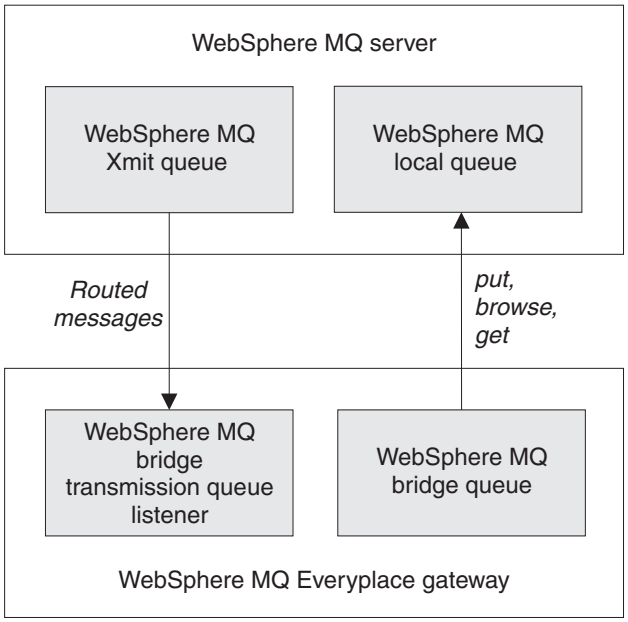

Figure 3. WebSphere MQ Everyplace interface to WebSphere MQ

Security

WebSphere MQ Everyplace includes an integrated set of security features that provide protection for message data, both 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 WebSphere MQ Everyplace messages while they are held by a local queue manager.

Queue-based security
Queue-based security automatically protects WebSphere MQ Everyplace message data between an initiating queue manager and a target queue, if 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 WebSphere MQ Everyplace application.

WebSphere MQ Everyplace security uses the authenticator, cryptor, and compressor attributes referred to in "WebSphere MQ Everyplace connections" on page 5. Queue based security is handled internally by WebSphere MQ Everyplace and does not require any specific action by the initiator or recipient of the message. Local and Message-level security must be initiated by an application.

WebSphere MQ Everyplace also provides a mini-certificate server for enhanced security.

See Chapter 8, "Security", on page 111 for detailed information about WebSphere MQ Everyplace security features.

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

Chapter 2. Getting started

The WebSphere MQ Everyplace C Bindings are a development environment for writing messaging and queuing applications in the C programming language. This section provides information to help you complete a successful installation of Version 2.0.0.5 of the C Bindings.

To successfully use the C Bindings you need to complete the following:

- Ensure that your system path is set to find the C Bindings DLLs, and is also set to the 'bin' directories of your JDK installation.
- Ensure that the C Bindings header files and library files are accessible to your compiler.
- Ensure that your application is compiled with the Microsoft Multithreaded DLL C Runtime.
- Ensure that the Windows Registry entries are created correctly.

These tasks are covered in more detail in the following sections.

Installation

To use the C Bindings you must have chosen to install the optional bindings code during your WebSphere MQ Everyplace installation.

The installed bindings consist of the following:

Binary files
These are DLLs and libraries that your applications need to link to. Your system needs to be set up so that your compiler can access these files. At runtime the DLL files need to be on the system path.

The DLL files are:

- HMQ_Bindings.dll
- HMQ_bindingsAPI.dll
- HMQ_bindingsOSA.dll
- HMQ_HAL.dll

There is also a static library:

- HMQ_bindingsCnst.lib

Header Files
These describe the C API and are installed in a directory called 'published'. The published directory must be in a path that is accessible to your C compiler.

Java Classes
Some classes that are required by the C Bindings are included with the main WebSphere MQ Everyplace classes for Java installation.

All the C Bindings files are under the `CBindings` directory in the main installation directory. The binary DLLs are in the `CBindings\bin` directory, the lib files are in `CBindings\lib` directory, and the header files are in the `include` directory under the main installation directory.

Setting the environment

Two options need to be specified to provide configuration information for the C Bindings. One specifies the location of options that are passed to the underlying Java VM, the other controls trace.

JVM options

You need to create a text file that contains options for the underlying JVM. The location of this can be specified either by a registry entry, or as an environment variable. The registry key takes precedence over the environment variable.

The registry key is `HKEY_LOCAL_MACHONE\SOFTWARE\IBM\MQe\CurrentVersion\Bindings` with a String value of `OptionsFile`.

The environment variable is `MQE_VM_OPTIONS_LOCN`. This variable points to the location of a text file that contains the options for the underlying JVM and it must be set.

In general, the JVM options can be specified on the Java command line, for example to set the classpath:

```
java -Djava.class.path=e:\myclasspath myClassToRun
```

The configuration file uses the same style to pass configuration options. Multiple options can be given to the JVM, and the configuration file should contain such options, one on each line. The following is an example configuration file:

```
#Example configuration file
 -Djava.class.path=e:\MQe;e:\MyApplicationClasses
 -Djava.compiler=NONE
```

- The first line is a comment, all comments should have # in the first column.
- The second line specifies the classpath. The classpath can contain spaces. double quotes do not need to be used.
- The third line indicates that the JIT should not be used. Do not set this compiler variable for normal operating conditions as it imposes an overhead on system performance.

Note: If you do not specify the above configuration options the JVM will not create, returning:

- `MQERETURN_JVM_SUPPORT`, Return code = 3000
- `MQEREASON_JNI_CREATE_VM_FAIL`, Reason code = 30003

Trace options

Trace in enabled with a registry key, `HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQe\CurrentVersion\Trace`.

All values are of type REG_SZ. The is the following two values are the minimum required for trace to be produced:

```
Enable=Yes
Location=c:\my\directory\structure\mqetrace
```

Where enable turns trace on and the location is the directory in which trace files are produced. This directory is created if it does not exist already.

In addition to these required values, there are a number of optional entries that affect the output trace.

Note: Enabling trace can produce very large files and, in general, you should only enable trace if instructed to by an IBM Support Representative.

JVM environment

The C Bindings APIs require a certified IBM or Sun Microsystems JVM v1.2.2, 1.3 or 1.4 and cannot be used with Personal Java or Java Micro Edition.

The WebSphere MQ Everyplace Version 2.0.0.5 C Bindings only support Windows platforms, that is NT, 2000, 98, 95, ME and XP.

Install the JVM as instructed in the chosen JDK, but make sure that the binaries for the installed JVM are available on the Windows system path. The `jre/bin` and `jre/bin/classic` directories in the JDK installation must also be included in the system path. However, if you are running JDK 1.4, include the `jre/bin` and `jre/bin/client` paths instead.

Compiling and Linking

A suitable C Compiler is required to build your application. Testing has been performed using the Microsoft Visual C++ v6 Environment.

Note: If C++ is used, ensure that the calling conventions are correctly maintained. All APIs use the __cdecl calling convention.

Your application needs to link against HMQ_bindingsAPI.lib and HMQ_bindingsCnst.lib. Linking against HMQ_bindingsCnst.lib is only required if you wish to use MQeString Constants. Also, in the Project Settings dialog:

- Select the **Link** tab
- Add HMQ_bindingsAPI.lib and HMQ_bindingsCnst.lib

The main header file that is required needs to be included as follows:

```
#include <published\MQe_API.h>
```

You also need to set a prepreocessor definition:

```
#define BINDINGS
```

You can either set the preprocessor definition in the compiler project settings or define it before including the MQe_API.h file. If this is not defined when the code is compiled, an error is returned.

Ensure that your compiler is set to enable this header file to be picked up. Note that the published directory prefix is required. This header file includes all the other headers that might be required. For example, if you are running Microsoft Visual C++, include the paths to the C Bindings header and library files as follows:

- Click **Tools —> Options**
- Select the **Directory** Tab
- Type the paths to the C Bindings header and library files

You need to set up the Windows System Path to point to the C Bindings DLLs from the Windows Control Panel or the Command Prompt.

Threading build time considerations

Consideration of the threading needs to be made both when setting up the environment to build your application and when designing the application. The build environment setup requires is described here, information on the application design considerations is provided in "Threading application design considerations" on page 19.

The underlying JVM has an implementation of threading. The C runtime used is selected to match the type of threading employed by the JVM. Under JDKs for the Windows platform, the Microsoft C Runtime is used, in particular the MultiThreaded DLL version. The debug version should be used for debug builds.

Failure to use the same Runtime as the JVM may result in application failures, typically with memory management due to different heaps being employed.

Relationship with the Native C Client

This version also includes the Native C Client, a full C implementation of WebSphere MQ Everyplace. While the C Bindings and the Native C Client share the API header files, there are some differences. These are documented in WebSphere MQ Everyplace C Programming Guide for Palm OS.

One difference is that, in the C Bindings, you must set the `#define BINDINGS` preprocessor definition before including the MQe_API.h header file. Also, the libraries that are linked have slightly different names.

The Native C Client and the C Bindings complement eachother, in that the Native C Client is for devices like Pocket PCs, while the C Bindings is supported on Windows platforms. The Native C Client does not include server functionality. Therefore, in a complete solution, it will need to use a server written using either using the Java code base or the C Bindings.

Examples

The following examples are provided to assist with the coding of WebSphere MQ Everyplace applications using the C Bindings.

Application examples

Ex1 Put and get a message

Ex2 Put an get multiple messages

Ex3 Message Listener

Ex4 Wait For a message

Ex5 Lock messages

Ex6 Assured put and get

Administration examples

Ex1 Create and delete a queue

Ex2 Add a connection definition

Ex3 Inquire about queue characteristics

Install Examples

Ex1 Create a queue manager

Ex2 Delete a queue manager

Ex3 Create a queue

Queue Manager Examples

Ex1 Client queue manager - this code is included in the Install Examples

Ex2 Server queue manager

Security Examples

Ex1 Client queue manager security example

Ex2 Server queue manager security example

Chapter 3. Using the C Bindings

This chapter introduces the philosophy behind the WebSphere MQ Everyplace C Bindings API, and outlines how to write an application using the Bindings.

Using handles

The WebSphere MQ Everyplace C API is *object* based, using handles to the objects as parameters. When an API function is called, you should always pass or receive a handle to an object. (There are a few exceptions but these are carefully noted in the WebSphere MQ Everyplace C Programming Reference.) The function called then acts on the object that is represented by the handle. The life cycle of the handles (and consequently the objects they represent) needs to be carefully managed. Functions are provided to create and free handles. Failure to free handles is equivalent to failure to free resources.

Handles are references to objects, and it is possible for handles to reference NULL objects. These handles should still be freed, as the handle itself uses resources (although these are minimal). A macro `IS_NULL(pObjectHandle)` is provided to determine if a handle points to a null object, as passing a handle to a null object into a API function can put the system into an unstable state.

API conventions

Most functions take a handle and an exception block. The handle indicate the object that the function is operating on. The exception block is specific to a thread and provides the ability to communicate errors back to the caller. Some functions do not take a handle. These are *Static* functions that do not operate on an object. Examples are the Session and terminate and initialization functions (discussed in "Session" on page 20). Functions that create new object handles do not take a handle as an input parameter, instead they return a new handle

All functions return `MQERETURN` values to indicate success or failure. Data is returned using output parameters, typically indicated by pointers to object handles or by pointers to primitive types.

Function prototypes

Prototypes follow two basic forms:

For the construction of new objects

```
MQERETURN mqeObjectName_new(MQeExceptBlock *pErrStruct,
                            MQeObjectHndl   *phNewObjectHndl,
                            .... <parameter>);
```

All other functions

```
MQERETURN mqeObjectName_functionName(MQeObjectHndl hObjectHndl,
                                     MQeExceptBlock *pErrStruct,
                                     ... <parameters>);
```

The handle of the object on which the function operates always comes first. With new functions, the exception block comes first as at that point there is no object to operate on.

For each **new** call there is a matching **free** call.

```
MQERETURN mqeObjectName_free
(MQeObjectHndl hObjectHndl, MQeExceptBlock  * pErrStruct);
```

Managing handles

To create a new object, and get the handle to that object, call the **mqeObjectName_new(...)** function. To release the resource associated with a handle, call the **mqeObjectName_free(...)** function.

When creating and freeing handles, the following rules should be observed:

- If a **new()** call fails, the handle does not need to be freed.
- Handles should not be freed more than once - doing so results in unpredictable behavior.
- You can free a `NULL` value and this does not cause an error.
- To check if a handle is NULL or represents a NULL object then use the `IS_NULL(pObjectHndl)`. This takes a pointer to an object handle and returns `MQE_TRUE` if it is NULL or represents NULL. Otherwise it return `MQE_FALSE`.

Note that for Administration messages, each message has its own unique **new()** function, but there is only one free function, **mqeAdminMsg_free**.

Object hierarchy

The underlying design of the C API is object based, and a hierarchy of objects can be established. The hierarchy reflects specialization of object types, an example is the Message object which is a specialization of the Fields object (seeChapter 4, “Fundamental objects”, on page 23).

As an example, assume that we have an MQeVehicle object and an MQeCar object. MQeCar is a specialization of MQeVehicle. Both of these objects could have the function **move()** defined on them, but MQeCar also has a **fillPetrol()** function. Both objects must also have **new()** and **free()** functions to create new objects and to return the handles.

Assume the following (using standard WebSphere MQ Everyplace API conventions):

```
mqeVehicle_new(MQeExceptBlock *pErrStruct, MQeVehicleHndl *phNewVehicleHndl);
mqeVehicle_free(MQeVehicleHndl hVehicleHndl, MQeExceptBlock *pErrStruct);
mqeVehicle_move(MQeVehichleHndl hVehicleHndl, MQeExceptBlock *pErrStruct);

mqeCar_new(MQeExceptBlock *pErrStruct, MQeCarHndl *phNewCarHndl);
mqeCar_free(MQeCarHndl hCarHndl, MQeExceptBlock *pErrStruct);
mqeCar_fillPetrol(MQeCarHndl hCarHndl, MQeExceptBlock *pErrStruct);
```

New objects can be created like this:

```
MQeVehicleHndl  hMyVehicle;
MQeCarHndl  hMyCar;

rc = mqeVehicle_new(&exceptBlock, &hMyVehicle);
rc = mqeCar_new(&exceptBlock, &MyCar);
```

Using the handle to the MQeCar object you can fill the car with petrol:

```
rc = mqeCar_fillPetrol(hMyCar, &exceptBlock);
```

Note that the following call would fail, with a Return Code `10400 - MQERETURN_INVLAID_ARGUMENT`, because MQeVehicle is not specialization of MQeCar.

```
rc = mqeCar_fillPetrol(hMyVehicle, &Block)
```

Both of the following calls are valid, because MQeCar is a specialization of MQeVehicle:

```
rc = mqeVehicle_move(hMyCar, &exceptBlock);
rc = mqeVehicle_move(hMyVehicle, &exceptBlock);
```

Such functions are known as polymorphic. Potentially different implements are called depending on the nature of handle you pass in.

To free the storage you should call the following functions

```
rc = mqeCar_free(hMyCar, &exceptBlock);
rc = mqeVehcile_free(hMyVehicle, &exceptBlock);
```

Static type checking

As discussed in the previous section, if object A is a specialization of object B, object A's handle can be used in functions defined on object B. When the application is built, the checks are made to see if the use of the handles is valid. Three levels of static type checking can be used:

- At the lowest level, no warnings are generated about mismatched handles.
- The middle level splits the objects up into sections. A number of predetermined *root* objects are defined. The handles for all specializations of a root object can be interchanged without warning, but use of a handle derived from another root object generates a compiler error.
- At the highest warning level, a compiler warning is generated for every handle use outside its own functions, even when the match is valid.

The middle warning level is the default, but this can be changed as follows:

Definition	Type checking level
`#define MQEHANDLE_WARNING_LEVEL 1`	No warnings at all
`#define MQEHANDLE_WARNING_LEVEL 2`	Default level
`#define MQEHANDLE_WARNING_LEVEL 3`	Maximum warnings

Using the example above, on the maximum warning level, the call

```
rc = mqeVehicle_move(hMyCar, &exceptBlock);
```

generates a compiler warning.

To remove this warning, convert the call to:

```
rc = mqeVehicle_move((MQeVehicleHndl)hMyCar, &exceptBlock);
```

On the lower two warning levels, this would not generate a warning.

The only overhead of changing the warning level is in the number of warnings produced during compilation.

Exception handling

Each function takes a pointer to an MQeExceptBlock object. All functions return an `MQERETURN` value. This value is used to indicate the general success or failure of a function. In addition each exception event generates a Reason Code which provides further information.

When using C code there are ***no*** exception handling mechanisms as in C++.

Note: The C Bindings do not interact with or use the operating system exception handling functionality

The structure of the exception block is:

```
struct MQeExceptBlockExtern_st {
   MQERETURN ec;
   MQEREASON erc;
   MQEINT32    dataArrayEntriedUsed;
   union {
      MQEINT32 index;
      MQEINT32 indexArr[MQE_EXCEPT_DATA_ARRAY_SIZE]
   } data;
}
```

The *dataArrayEnties* indicates the number of entries used in the data union. This is used in one of the following ways.

- The API functions perform a limited check on the parameters that are passed into them. If a parameters is wrong in some regard, the index indicates which parameters is in error.
- For other exceptions, the *indexArr* contains a "stack trace" of exceptions. These indicate the original exception that occurred and other specializations of the exception. This exception stack should not be used under normal circumstances; it should be used only in Service situations when a problem has been reported.

Full details of the return codes and reason codes are provided in the WebSphere MQ Everyplace Application Programming Guide under the MQe_ReturnCodes.h header file. Please see the example code for examples of how the Exception Block is used.

Obtaining an Exception Block

You are recommended to allocate the exception block on the stack, rather than the heap. This is to simplify possible memory allocations, although there is no programmatic restriction on allocating space on the heap.

In its simplest form your code would look like this:

```
MQERETURN rc
MQeExceptBlock exceptBlock;
/* .... initialisation */
rc = mqeFunction_anyFunction(&exceptBlock,   /* parameters go here */ );
if (MQERETURN_OK != rc ) {
printf("An error has occured , return code = %d,  reason code = %d \n",
           exceptBlock.ec exceptBlock.erc);
} else {
}
```

If you are not interested in the status of a call, NULL can be passed as the Exception Block.

Note: If an error does occur and corrective action is not taken subsequent API calls could put the system in an unpredictable state.

Unsupported Java APIs

The following classes that are included in the Java codebase are not supported by the C bindings:

- MQeMiniCertIssuanceInterface
- MQeTransformerInterface

- MQeMQBridgeQueue
- All Rules classes
- All Adapter classes
- MQeRunListInterface
- MQeEventLogInterface
- MQeTraceInterface
- MQeSecurityInterface

Subclassing of existing Java classes is not be supported by the C Bindings.

WebSphere MQ Everyplace rules, adapters, and transformers cannot be coded in C, If you wish to use any of these functions they must be coded in Java.

Information about rules, adapters, and transformers, and how to use them can be found in *WebSphere MQ Everyplace Application Programming Guide* and *WebSphere MQ Everyplace Programming Reference*.

Use of Java class names

When configuring a number of the features of WebSphere MQ Everyplace, you need to pass a string that represents the name of the class to load. This is simpler if you use an alias. If you wish to customize behavior in a given circumstance, you can pass the name of your own class. The location of your classes need to be added to the classpath in the configuration file.

Type mapping

When exchanging messages between a Java queue manager and a C queue manager, the following type mappings are used:

C Bindings type	Java type
MQEBOOL	boolean
MQEBYTE	byte
MQECHAR16	char
MQEINT16	short
MQEINT32	int
MQEINT64	long
MQEFLOAT	float
MQEDOUBLE	double

The range of types available in C is greater than in Java, but the use of types that have no mapping is not supported.

Threading application design considerations

Threading needs to be considered both at build time and at run time. The application design considerations are described here, the build time requirements are described in “Threading build time considerations” on page 12.

Run time

The session initialize should be performed before any other threads have started. All threads should finish before the session terminate is called; this includes all threads that might be involved in callback style operations.

Due to the underlying Java VM, the same threading model needs to be used in the application. In the case of the Sun and IBM JDKs this means that native Win32 threads should be used. In addition these JDKs used the MSVCRT C Runtime library, and this should be used in the application.

When a new thread wishes to access any WebSphere MQ Everyplace API function it must call the **threadAttach** function (found in the session API). When the thread has finished using the WebSphere MQ Everyplace API (this could be well before the thread terminates) it should call the **threadDetach** function.

Failure to call the **threadDetach** function prior to the thread terminated (either naturally or forced) results in the system being put into an unstable state. This could result in premature failure.

Notes:

1. If the number of calls to **threadAttach** and **threadDetach** do not match, **sessionTerminate** does not complete successfully.
2. Do not call **threadAttach** multiple times on the same thread without calling **theadDetach** first. There is no restriction on calling multiple **threadAttach**/**threadDetach** pairs. Depending on the nature of your application it may be profitable to call **theadDetach** if your thread is going to do significant non-WebSphere MQ Everyplace work. Profiling of the particular application is recommend to determine whether this is worthwhile.

Session

A *session* is defined as the time between the unitization of the MQSeries Everyplace system, and its termination, within a single memory space. In terms of windows there is one session for each processes. With the underlying implementation being in Java , this implies that a session has only one queue manager.

The session should be initialized and then terminated before the process exits. While session initialization and termination are being performed, no other thread should access the MQSeries Everyplace API. Failure to follow these procedures results in unpredictable errors. After termination any thread accessing the API receives an error.

Session unitize and terminate should be called on the same thread.

API calls

Full details of the session initialize and terminate calls are provided in the *MQSeries Everyplace C Programming Guide*. Both calls take an exception block.

Administration messages

In the Java code, administration messages form a class hierarchy. The base class MQeMsgObject is extended by MQeAdminMsg, from which many other administration messages are derived. The derived administration messages often use the functions that they inherit from previous levels in the object hierarchy.

In the C API a similar mechanism can be used. For example, an MQeAdminMsg can be created, and a handle returned to it. This handle can then be used to call the functions of MQeMsgObject of which it is a specialization. Similarly an MQeClientConnectionAdminMsg (a specialization of MQeMsgObject) handle can be used on an MQeAdminMsg function.

WebSphere MQ bridge

Under Java, the WebSphere MQ bridge is constructed from a number of different objects. Some of these are created from published classes and others are created using administration messages. The objects that have a published Java class have a corresponding interface in C.

When constructing administration messages to create WebSphere MQ bridge objects, the full Java class name must be specified (see "Administration of the WebSphere MQ bridge" on page 89).

Chapter 4. Fundamental objects

There are a number of objects within WebSphere MQ Everyplace that are fundamental to the system. These will be explained in this section. The main objects are:

MQeString
A representation of character strings used by the API, providing full support for Unicode

MQeFields
An object used to hold data and information for processing messages

MQeString, and MQeFields are both described in this section

MQeString

The MQeString class contains user defined and System Strings and is an abstraction of character strings. It allows a string to be created or retrieved in a number of formats such as arrays containing Unicode code points (each code point stored in a 1, 2, or 4 bytes memory space) and UTF-8. The current implementation only supports these external formats.

MQeString is used throughout the C-API where a string is required.

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

Constant Strings

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

- MQe_Admin_Constants.h
- MQe_Connection_Constants.h
- MQe_MQBridge_Constants.h
- MQe_MQe_Constants.h
- MQe_Queue_Constants.h
- MQe_Registry_Constants.h

Constructor

There are different string constructors that are represented as CHAR8, CHAR16, CHAR32, and UTF8:

-
```
MQERETURN mqeString_newChar8(MQeExceptBlock *pErrStruct,
            MQeStringHndl *phNewString,
            MQECONST MQECHAR *pInput)
```
-
```
MQERETURN mqeString_newChar16(MQeExceptBlock *pErrStruct,
            MQeStringHndl *phNewString,
            MQECONST MQECHAR16 *pInput)
```
-
```
MQERETURN mqeString_newChar32(MQeExceptBlock *pErrStruct,
            MQeStringHndl*phNewString,
            MQECONST MQECHAR32 *pInput)
```
-
```
MQERETURN mqeString_newUtf8(MQeExceptBlock *pErrStruct,
           MQeStringHndl *phNewString,
           MQECONST MQECHAR *pInput)
```

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

Destructor

```
MQERETURN mqeString_free(MQeStringHndl hString,
        MQeExceptBlock *pErrStruct)
```

This function destroys an MQeString object that was created using **mqeString_new**, or returned by any other function.

Getter These functions populate a character buffer with the contents of an MQeString performing conversion wherever necessary. Only simple conversions are carried out. No codepage conversion is attempted. For example, if an SBCS string has been put into the string, then trying to get the data out as DBCS (Unicode) data works correctly. However, if the data was put in as DBCS, and you try to get the data out as SBCS, this only works if the data does not have any values that cannot be represented with a single byte. When **get()** is used for SBCS, DBCS, or QBCS, each character is represented by its Unicode code point value. The caller of these functions are expected to allocate the memory. If the pOutputString pointer is null, or the size (pointed to by pSize) is less than zero, the required size of the buffer in the number of characters will be returned.

-
```
MQERETURN mqeString_newChar8(MQeExceptBlock *pErrStruct,
            MQeStringHndl *phNewString,
            MQECONST MQECHAR *pInput)
```
-
```
MQERETURN mqeString_newChar16(MQeExceptBlock *pErrStruct,
            MQeStringHndl *phNewString,
            MQECONST MQECHAR16 *pInput)
```
-
```
MQERETURN mqeString_newChar32(MQeExceptBlock *pErrStruct,
            MQeStringHndl *phNewString,
            MQECONST MQECHAR32 *pInput)
```
-
```
MQERETURN mqeString_newUtf8(MQeExceptBlock *pErrStruct,
            MQeStringHndl*phNewString,
            MQECONST MQECHAR *pInput)
```

Tester

-
```
 MQERETURN  mqeString_isAsciiOnly(MQeStringHnld hString,MQeExceptBlock
             *pErrStruct,MQEBOOL *pisAsciiOnly)
```

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

-
```
MQERETURN mqeString_equalTo(MQeStringHndl hString,MQeExceptBlock
          *pErrStruct,MQEBOOL *pIsEqual,
          MQECONST MQeStringHndl hEqualToString)
```

  This function determines whether two strings are equivalent.

-
```
MQERETURN mqeString_isNull(MQeStringHndl hString, MQeExceptBlock
          *pErrStruct,MQEBOOL *pIsNull)
```

  This function determines if a string is a null string (a NULL handle is also considered as a null string).

-

```
MQERETURN mqeString_codePOintSize(MQeStringHndl hString, MQeExceptBlock
           *pErrStruct,MQEINT32 *pSize)
```

This function finds the number of bytes required to store the largest code point in the string.

MQeFields

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

```
MQeFieldsHndl hNewFields;
 MQERETURN rc;
 rc = mqeFields_new(&errStruct, &hNewFields);
```

There are various **put** and **get** functions 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

Note: Java 1.4 and earlier versions, are not capable of handling Unicode codepoints larger than six charaters. As a result, bindings cannot put unicode strings containing chars larger than two bytes successfully, although Java will not complain.

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:

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

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

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

Arrays of values may be held within a fields object. There are two forms for holding arrays, *fixed length*, and *variable length*. A function of the form shown in the following example is used to store the arrays. The boolean option is set to `MQE_TRUE` for variable length.

```
MQERETURN mqeFields_putDoubles(MQeFieldsHndl hFields,
      MQeExceptBlock * pErrStruct,
      MQECONST MQeStringHndl hFieldName,
      MQECONST MQEDOUBLE * pDoubles,
      MQEINT32 arrLength,
      MQEBOOL multiFieldsRep);
```

Bytes are a special case, as they can handle multi-dimensional arrays.

For one byte

```
MQERETURN mqeFields_putByte( MQeFieldsHndl hFields,
     MQeExceptBlock * pErrStruct,
     MQECONST MQeStringHndl hFieldName,
     MQEBYTE input);
```

For a 1-dimensional array of bytes

In effect this is a fixed length array.

```
MQERETURN mqeFields_putBytes(MQeFieldsHndl hFields,
     MQeExceptBlock * pErrStruct,
     MQECONST MQeStringHndl hFieldName,
     MQECONST MQEBYTE * pBytes,
     MQEINT32 arrLength);
```

For a 2-dimensional array of bytes

This is in effect a variable length array.

```
MQERETURN mqeFields_putByteArray(MQeFieldsHndl hFields,
      MQeExceptBlock * pErrStruct,
      MQECONST MQeStringHndl hFieldName,
      MQECONST MQEBYTE ** ppBytes,
      MQECONST MQEINT32 * pLenOfByteSeqs,
      MQEINT32 arrLength);
```

To retrieve arrays, get functions can be used as required. There is only one form: `mqeFields_getXXXs()`

Chapter 5. Queue managers, messages, and queues

Chapter 1, "Overview", on page 1 provides a high level description of the services provided by WebSphere MQ 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 34)
- A queue manager definition
- Local default queue definitions (see "Queues" on page 41)

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.

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

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

Queue manager names

WebSphere MQ Everyplace queue manager names can contain the following characters:

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

Queue manager names cannot have a leading or trailing '.' character.

There are no inherent name length limitations in WebSphere MQ Everyplace.

For additional naming recommendations when interacting with WebSphere MQ networks, see "Naming recommendations for inter-operability with a WebSphere MQ network" on page 80.

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 create the MQeQueueManagerConfigure object by calling the **mqeQueueManagerConfigure_new** function. Apart from the *ExceptionBlock* and the new MQeQueueManagerConfigure *Handle*, this function takes two additional parameters.

The method of operation depends on these parameters. "NULL" can be passed for these parameters, in which case **mqeQueueManagerConfigure_activate** is called immediately after **mqeQueueManagerConfigure_new**. Alternatively startup parameters can be passed.

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

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

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

The example code includes creating an instance of MQeQueueManagerConfigure.

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 **mqeQueueManagerConfigure_setDescription()**
- You can set a connection time-out value with **mqeQueueManagerConfigure_setChannelTimeout()**
- You can set the name of the connection attribute rule with **mqeQueueManagerConfigure_setChnlAttributeRuleName()**

Call **mqeQueueManagerConfigure_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.

At this point you can **close()** and **free()** 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:
  mqeQueueManagerConfigure_defineDefaultAdminQueue()
- An administration reply queue:
  mqeQueueManagerConfigure_defineDefaultAdminReplyQueue()
- A dead letter queue:
  mqeQueueManagerConfigure_defineDefaultDeadLetterQueue()
- A default local queue:
  mqeQueueManagerConfigure_defineDefaultSystemQueue()

All these functions return an error 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 to store messages that cannot be delivered to their correct destination.

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

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. Once the **close()** function has been completed, the handle to the MQeQueueManagerCondigure must to be freed by calling the **free()** function

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 28.

3. Delete the standard queue and queue manager definitions

Delete the default queues by calling:

- **mqeQueueManagerConfigure_deleteAdminQueueDefinition()** to delete the administration queue
- **mqeQueueManagerConfigure_deleteAdminReplyQueueDefinition()** to delete the administration reply queue
- **mqeQueueManagerConfigure_deleteDeadLetterQueueDefinition()** to delete the dead letter queue
- **mqeQueueManagerConfigure_deleteSystemQueueDefinition()** to delete the default local queue

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

Delete the queue manager definition by calling **mqeQueueManagerConfigure_deleteQueueManagerDefinition()**

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

4. Close the MQeQueueManagerConfigure instance

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

Using queue manager aliases

Aliases can be used for WebSphere MQ Everyplace queue managers, and can be used by application programs, to provide a level of indirection between the application and the real object. Hence the attributes of a queue manager that an alias relates to can be changed without the application needing to change.

The following examples illustrate some of the ways that aliasing can be used with queue managers.

Examples of queue manager aliasing

Addressing a queue manager with several different names

Suppose you have a queue manager `SERVER23QM` on the server `SAMPLEHOST`, listening on port `8082`. You have an application `SERVICEX` that accesses this queue manager, and wants to refer to the queue manager as `SERVICEXQM`. This can be achieved using an alias for the queue manager as follows:

- **Configure a connection on the SERVER23QM :**

  Connection Name/Target queue manager: `SERVICEXQM`

  Description: Alias definition to enable `SERVER23QM` to receive messages sent to `SERVICEXQM`

  Channel: "null"

  Network Adapter: "null"

Network adapter options: "null"

- **Create a local queue on the SERVER23QM queue manager:**

  Queue Name: SERVICEXQ

  Queue Manager: SERVER23QM

  The server-side application takes messages from this queue, and process them, sending messages back to the client.

An WebSphere MQ Everyplace application running within the server's JVM can now put messages to the SERVICEXQ on either the SERVER23QM queue manager, or the SERVICEXQM queue manager. In either case, the message will arrive on the SERVICEXQ.

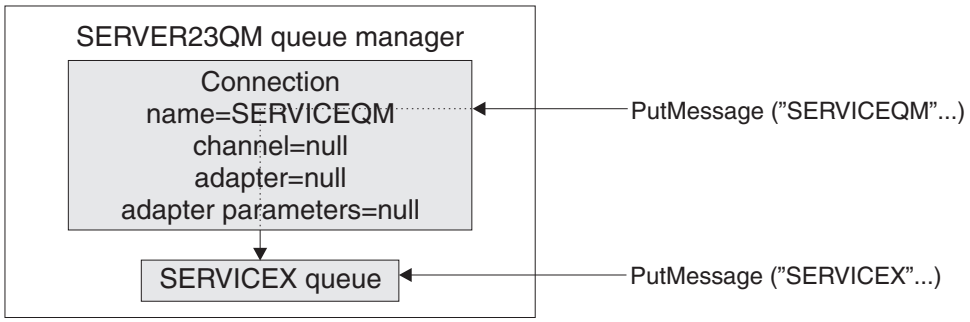

Figure 4. Addressing a queue manager with two different names

If the SERVICEXQ queue is moved to another queue manager, the connection alias can be set up on the new queue manager, and the applications do not need to be changed.

Different routings from one queue manager to another

Using the scenario just described, an WebSphere MQ Everyplace queue manager on a mobile device (MOBILE0058QM) can now access the SERVICEXQ queue in a number of different ways. Two examples are described here:

- **Aliasing on the sending side**

  Using this method of routing, the receiving queue manager does not know that the sending queue manager has given him an alias name. The aliasing is confined to the sending queue manager only.

  On the mobile device:

  – Create a connection from MOBILE0058QM to the SERVER23QM queue manager:

    Connection name
    SERVER23QM

    Network Adapter parameter
    Network:SAMPLEHOST:8082

  – Create an alias called SERVICEXQM for queue manager SERVER23QM

  When a message is sent from the mobile device application to the SERVICEXQM queue manager, WebSphere MQ Everyplace maps the SERVICEXQM name to SERVER23QM in the connection , and sends the message to the SERVER23QM queue manager.

If the Mobile58QM then wished to send its messages to a different server queue manager, Server24QM, it would remove the alias SERVICEXQM from the Server23QM connection, and add it to a Server24QM connection. This has no impact on the receiving queue managers, or the sending applications.

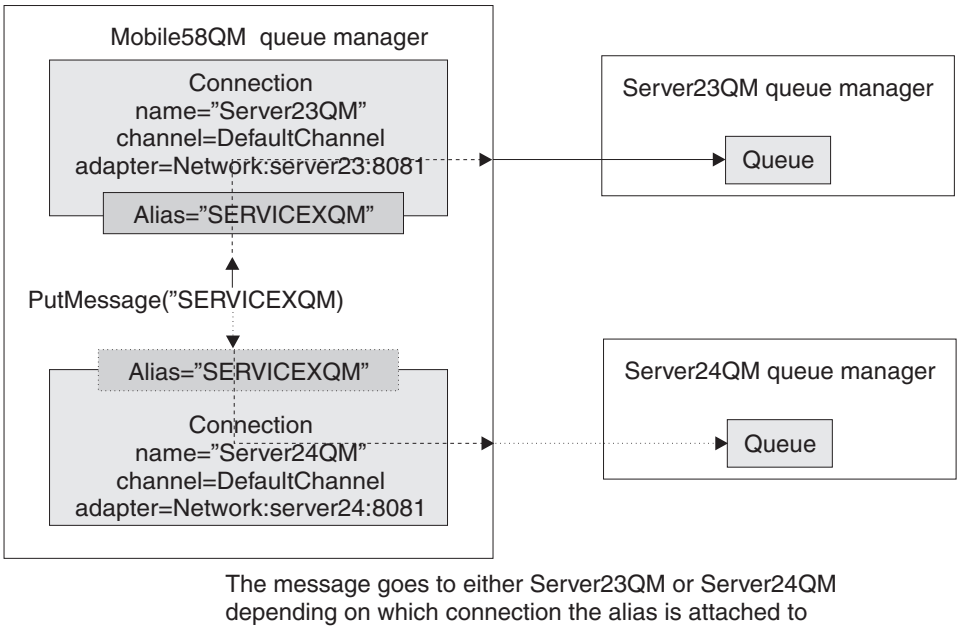

Figure 5. Addressing a queue manager with two different names

- **Virtual queue manager on the receiving side**

  Using this method, the sending queue managers think that its messages are routed through an intermediate queue manager before reaching the target queue manager. The target queue manager doesn't actually exist. The 'intermediate' queue manager captures all the message traffic for this virtual target queue manager.

  On the mobile device:

  – Create a connection from MOBILE0058QM to the SERVER23QM queue manager:

    Connection name SERVER23QM

    Network Adapter parameter Network:SAMPLEHOST:8082

  – Create a second connection to the SERVICEXQM that routes messages through the first connection:

    Connection name
    SERVICEXQM

    Network Adapter parameter
    SERVER23QM

    Note: This is not an alias. It is a *via routing*, indicating that messages headed for SERVICEXQM are to be routed via the SERVER23QM queue manager on the receiving side.

  The via routing on the mobile device causes any messages that are put to SERVICEXQM to be directed to Server23QM. Server23QM gets the

messages and notes that they are destined for the SERVICEXQM queue manager. It resolves the SERVICEXQM name and finds that it is an alias which represents the Server23QM queue manager (itself). The Server23QM queue manager then accepts the messages and puts them onto the queue.

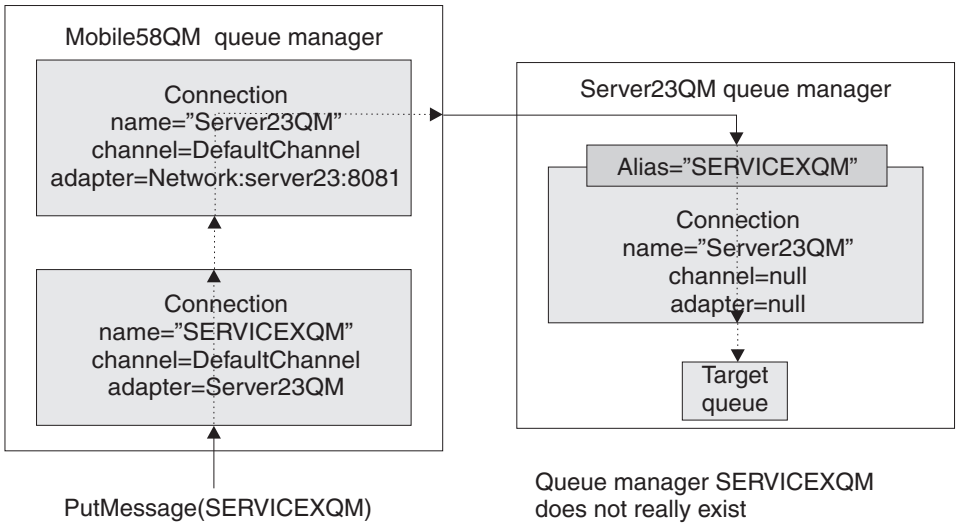

Figure 6. Addressing a queue manager with two different names

As an alternative to the above, you can keep the SERVICEXQM in existence, but move it from its original machine to the same machine (but a different JVM) as the Server23QM queue manager. SERVICEXQM needs to listen on a different port, so the connection from Server23QM to SERVICEXQM needs to be changed as well.

Starting queue managers

A queue manager can run:

- as a client
- in a server

The following sections refer extensively to the example code to illustrate how to start queue managers. All queue managers are constructed from the same base WebSphere MQ Everyplace components, with some additions that give each its unique properties. WebSphere MQ Everyplace provides a class MQeQueueManagerUtils that encapsulates many of the common functions.

All the examples require parameters at startup.

Client queue managers

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

Class Aliases

The aliases are not processed by the queue manager itself. The queue manager requires these aliases 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 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:

MQE_REGISTRY_LOCAL_REG_TYPE
: The type of registry being opened. *file registry* and *private registry* are currently supported. A private registry is required for some of the security features. See Chapter 8, "Security", on page 111.

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:

MQE_REGISTRY_DIR_NAME
: The name of the directory holding the registry files.

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

MQE_REGISTRY_DIR_NAME
: The name of the directory holding the registry files

MQE_REGISTRY_PIN
: The PIN for the private registry

MQE_REGISTRY_KEY_RING_PASSWORD
: The password or phrase used to protect the registry's private key

MQE_REGISTRY_CA_IP_ADDR_PORT
: The address and port number of a mini-certificate server

MQE_REGISTRY_CERT_REQ_PIN
: 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 server.

Note: For security reasons, the *PIN* and *KeyRingPassword*, if supplied, are deleted from the startup parameters as soon as the queue manager has been activated.

For either type of registry *MQE_REGISTRY_SEPARATOR* 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 "+".

RegistryAdapter:

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

Starting a client queue manager

Starting a client queue manager involves:

1. Adding any aliases to the system
2. Enabling trace if required
3. Starting the queue manager

The following code fragment starts a client queue manager:

```
MQERETURN createQueueManager(MQeExceptBlock *pErrorBlock, MQeQueueManagerHndl
          *phQMgr, MQeFieldsHndl hInitFields,
          MQeStringHndl hQStore)
{

   MQERETURN rc;
   MQeQueueManagerConfigureHndl hQMgrConfigure;

      /* Create instance of QueueManagerConfigure Class */
      rc = mqeQueueManagerConfigure_new(pErrorBlock,&hQMgrConfigure,
              hInitFields,hQStore);

      if (MQERETURN_OK == rc) {
         /* define queue manager */
         rc = mqeQueueManagerConfigure_defineQueueManager(hQMgrConfigure,
                    pErrorBlock);
         if (MQERETURN_OK == rc) {
            /* define system default queues */
            rc = mqeQueueManagerConfigure_defineDefaultSystemQueue
              (hQMgrConfigure, pErrorBlock, NULL);
         }

         /* close mqeQueueManagerConfigure */
         (void)mqeQueueManagerConfigure_close(hQMgrConfigure, NULL);
         if (MQERETURN_OK == rc) {
            /* create queue manager */
            rc = mqeQueueManager_new(pErrorBlock, phQMgr);
            if (MQERETURN_OK == rc) {
```

```
                rc = mqeQueueManager_activate(*phQMgr, pErrorBlock, hInitFields);
              }
            }
            /* free mqeQueueManagerConfigure */
            (void)mqeQueueManagerConfigure_free(hQMgrConfigure, NULL);
          }

   return rc;
}
```

Once you have started the client, you can obtain a reference to the queue manager object by using API call **mqeQueueManager_getReference(queueManagerName)**.

Server queue managers

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

Example MQeServer

MQeServer is the simplest server implementation.

This server can be started with the following command:

```
qm_server server_QMgr_name [-p private_reg_PIN]
```

You must supply the *-p* parameter if the queue manager uses a private registry. Otherwise, the queue manager's registry is treated as a file registry. The program activates the queue manager (including a channel listener listening on port 8081) and goes into an indefinite sleep.

Use `ctrl-C` to shut down the server.

To delete the constructed queue manager, use the example qm_delete.

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

The new parameters control the use of the connection. The *MaxChannels* parameter controls the maximum number of connections that can be open at any time. A special value of `0` means that the queue manager can handle an unlimited number of connections.

The following parameters control 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::8082` means use the `Network` adapter where `Network` is an alias to listen on port 8082. (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 connections are timed out. As connections 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 connections that have been inactive for a period of time.

Once the server has been initialized it must be activated.

When you activate a server the following occurs:

1. A channel manager is started
2. The queue manager is started
3. The channel listener is started

Code to demonstrate server activation is provided in queue manager example Ex2.

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 111.

Messages

WebSphere MQ Everyplace messages are descendant objects of MQeFields, as described in Chapter 4, “Fundamental objects”, on page 23. Applications can put data into the message as a <name, data> pairing.

MQeMessages are specializations of MQeFields objects, therefore the functions that are applicable to MQeFields can be used with MQeMessage. An example of creating a message plus adding a fields is shown below. This is a function from one of the examples:

```
MQERETURN creatAMessage(MQeExceptBlock *pErrorBlock, MQeMsgHndl * phOutMsg) {

   MQERETURN rc;

   /* create a message obj */
   rc = mqeMsg_new(pErrorBlock, phOutMsg);

   if (MQERETURN_OK == rc) {
      MQeStringHndl hDataFieldName;

      rc = mqeString_newChar8(pErrorBlock, &DataFieldName, "myData");
      if (MQERETURN_OK == rc) {
         /* put some data in */
         rc = mqeFields_putInt32((MQeFieldsHndl)*phOutMsg, pErrorBlock,
            hDataFieldName, 1);
      } else {
         displayError("new MQeString error (for data field name)", pErrorBlock);
      }
   } else {
      displayError("new MQeMsg error", pErrorBlock);
   }
   return rc;
}
```

This shows a message being created, a new field name being created, and an integer being placed into the field. The field can be retrieved using the **mqeField_getXXX** functions.

WebSphere MQ Everyplace defines some constant field names that are useful to messaging applications. These are:

Unique ID
MQE_MSG_ORIGIN_QMGR + MQE_MSG_TIME

Message ID
MQE_MSG_MSGID

Correlation ID
MQE_MSG_CORRELID

Priority
MQE_MSG_PRIORITY

The *Unique ID* is a combination of a unique (per process) 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 WebSphere MQ Everyplace network so long as all queue managers within the WebSphere MQ Everyplace network are named uniquely.

Note: WebSphere MQ 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 **mqeMsg_getMsgUIDFields()** function accesses the *Unique ID* of a message:

The **mqeMsg_getMsgUIDFields()** function returns an MQeFields object that contains two fields,

- MQe.Msg_OriginQMgr
- MQe.Msg_Time

These fields can be individually retrieved as follows:

The WebSphere MQ *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 WebSphere MQ family.

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

Storing messages

Most queue types hold messages in a persistent store. While in the store, the state of the message varies as it is transferred into and out of the store. As shown in Figure 7 on page 39:

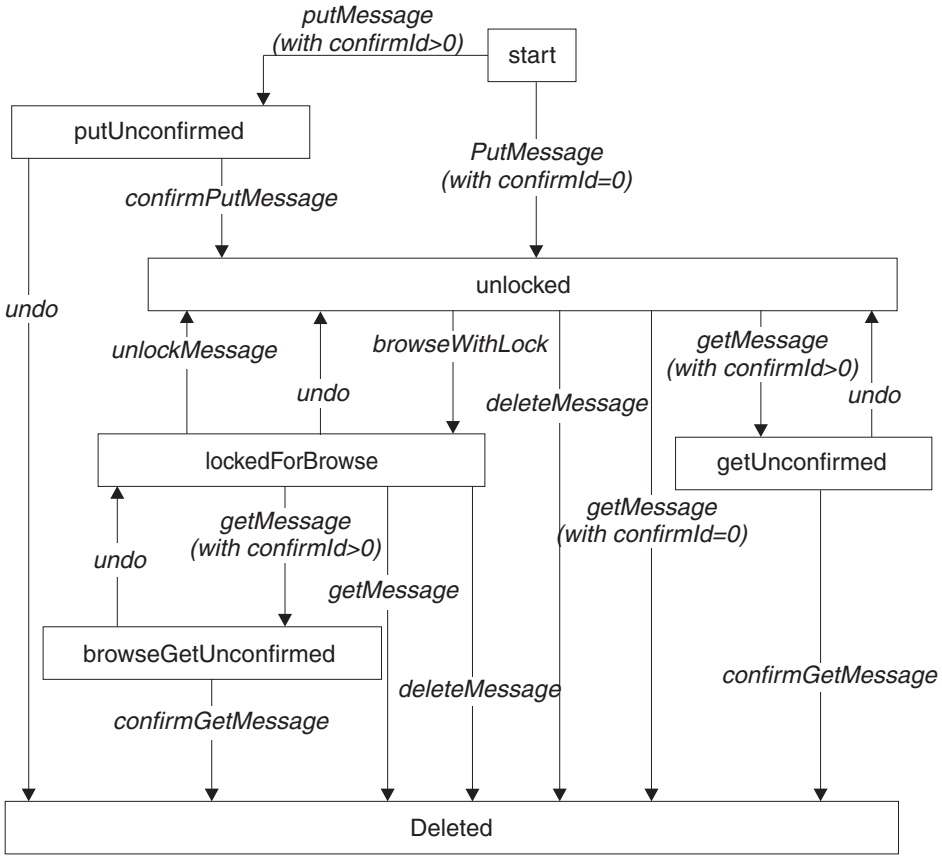

Figure 7. Stored message state flow

Message states

The possible message states are:

Start The initial state of a message before it is added to the message store.

Put Unconfirmed
A message has been placed in the message store under a *confirmID* but its addition has not been confirmed. The message is effectively hidden from all actions except **confirmPutMessage**, **confirm**, or **undo**.

Unlocked
A message has been added to the message store. There is no lock on it, and it is visible to all queries.

Locked for Browse
A browse with lock has retrieved the message. The message is now hidden from all queries except **getMessage**, **unlockMessage**, and **undo**.

Get Unconfirmed
A get message has been made with a *confirmID* but the get has not been confirmed. The message is invisible to all queries except **confirmGetMessage**, **confirm**, or **undo**. Each of these actions requires the matching *confirmID* to be included to confirm the get.

Browse Get Unconfirmed
A message has been got while it is locked for browse. This can only be done by passing the correct *lockID* to the **getMessage** function.

Deleted
The final state, after a message has been removed from the database.

Message events

Messages pass from one state to another as a result of an event. The possible message events (as shown in Figure 7 on page 39) are:

putMessage
Message placed on message store, no confirm required.

getMessage
Message retrieved from message store, no confirm required.

putMessage with confirmId>0
Message placed on message store, confirm required.

confirmPutMessage
A confirm for an earlier **putMessage** with confirmId>0.

getMessage with confirmId>0
Message retrieved from message store, confirm required.

confirmGetMessage
A confirm for an earlier **getMessage** with confirmId>0.

browseWithLock
Browse messages and lock those that match. Prevents messages changing while browse is in operation.

unlockMessage
Unlock a message locked with a **browsewithLock** command.

undo Unlock a message locked with a browse, or undo a **getMessage** with confirmId>0 or **putMessage** with confirmId>0.

deleteMessage
Remove a message from the message store.

More detailed descriptions of message events and states are included in "Assured message delivery" on page 47, and "Browse and Lock" on page 42 and

Message index fields

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

Unique ID
`MQE_MSG_ORIGIN_QMGR + MQE_MSG_TIME`

Message ID
MQE_MSG_MSGID

Correlation ID
MQE_MSG_CORRELID

Priority
MQE_MSG_PRIORITY

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

Filters

The concept of filters allows WebSphere MQ Everyplace to perform powerful message searches. Most of the major queue manager operations support the use of filters. The **mqeQueueManager_getMessage** function has an *MQeFields* parameter. This Fields object is the filter. If you pass "null" in then no filter is used.

The use of a filter causes an application to return the first available message that contains the same fields and values as the filter. An example of using a filter on a **getMessage()** function is in Application example Ex2.

When a filter is applied to a search, the fields in the filter are compared with each index entry in turn. If a field is common to both the index entry and the filter, and the values in the field are different, then the message cannot possibly match the filter and it is excluded from consideration. If a field is not common to both filter and index entry, or if the field is common and the values are the same, then the message is included in the search.

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.

Messages can also have an expiry interval that overrides the value of any queue expiry interval. 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).

To set a relative expiry time use the following on a message handle:

```
mqeFields_putInt32(pErrorBlock, hMsg, relativeTime);
```

To set an absolute expiry time use:

```
mqeFields_putInt64(pErrorBlock,hMsg, absoluteTime);
```

All Times are in milliseconds

Queues

Queue managers manage queues and provide programming interface access to the queues. The queues are not directly visible to an application and all interactions with the queues take place through queue managers. Each queue manager can have queues that it manages and owns. These queues are known as *local* queues. WebSphere MQ 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 43).

The messages on the queues are held in the queue’s persistent store (see “Storing messages” on page 38). The backing store used by a queue can be changed using an WebSphere MQ 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 WebSphere MQ Everyplace to assure the delivery of messages.

Queue names

WebSphere MQ Everyplace queue names can contain the following characters:

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

Queue names cannot have a leading or trailing '.' character.

There are no inherent name length limitations in WebSphere MQ Everyplace.

For additional naming recommendations when interacting with WebSphere MQ networks, see "Naming recommendations for inter-operability with a WebSphere MQ network" on page 80.

Queue types

The WebSphere MQ Everyplace queue types are described briefly in "WebSphere MQ Everyplace queues" on page 3, and information on setting up and administering the various types is provided in "Queues" on page 66.

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 returns `Return Code 121 - MQERETURN_J_Q_NO_MATCHING_MSG` if a **get** message command is issued. This allows you to create an application that reads all the available messages on a queue.

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.The example program Example 5 contains an example of browsing messages with lock.

This command locks all the messages on the local queue `SYSTEM.DEFAULT.QUEUE.NAME`. 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 MQeMessageEnumeration object contains all the messages that match the filter supplied to the browse. .

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.

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

Note: See "Getting and browsing messages from the WebSphere MQ bridge queue" on page 106 for special considerations with WebSphere MQ bridge queues.

Message listeners

WebSphere MQ Everyplace allows an application to listen for events occurring on queues. The application is able to specify message filters to identify the messages in which it is interested. A message arriving on queue triggers an event.

The Application example Ex 4 contains an example of using message listeners.

To enable this, an MQeMessageListener object must be created. The **new()** function for this takes an extra parameter, which is a function pointer. This function pointer indicates the function that should be called when a message arrives on the queue. This function should have the prototype:

```
- MQEVOID messageArrived(MQeMessageEventHndl hE);
```

Where `MQeMessageEventHndl` is a handle to an object that contains information about the message that has arrived. The queue needs to be aware of this handle , so the **mqeQueueManager_addMessageListener** function must be called with the MQeMessageListener object handle.

The MQeMessageEvent contains information about the message including:

- The name of the queue on which the message arrived
- The UID of the message
- The Correlation ID of the message
- The Message Priority

Message filters only work on local queues. A separate technique known as message polling allows messages to be obtained as soon as they arrive on remote queues. (This is discussed in the next section.)

Message polling

Message polling uses the **mqeQueueManager_waitForMessage()** function. This command issues a **mqeQueueManager_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:

```
rc = mqeQueueManager_waitForMessage(hQueueManger, pExceptBlock, &hMQeMessage,
                                    hQMgrName, hQueueName, hFilter,
                                    hAttribute, confirmID, 6000);
```

The **mqeQueueManager_waitForMessage()** function 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 6 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 1 shows the operations that can be performed on messages on the various queue types.

Table 1. 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	

Using queue aliases

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

The following examples illustrate some of the ways that aliasing can be used with queues.

Examples of queue aliasing

Merging applications

Suppose you have the following configuration:

- A client application that puts data to queue `Q1`
- A server application that takes data from `Q1` for processing
- A client application that puts data to queue `Q2`
- A server application which takes data from `Q2` for processing

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

Upgrading applications

Suppose you have the following configuration:

- A queue `Q1`

- An application that gets messages from Q1
- An application that puts messages to Q1

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

Using different transfer modes to a single queue

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

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

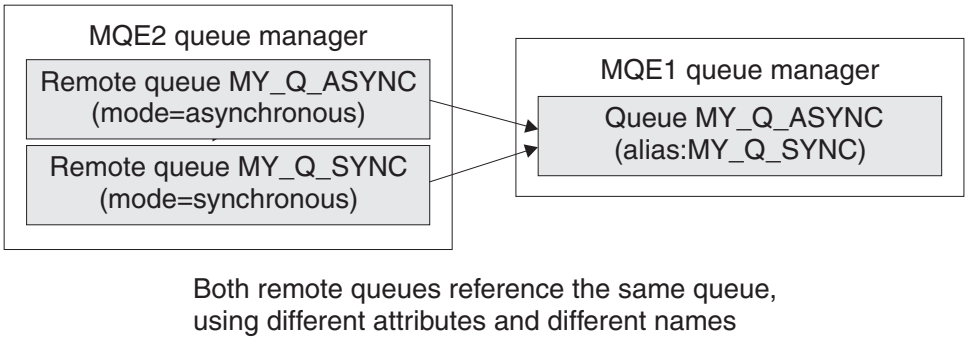

Figure 8. Two modes of transfer to a single queue

Synchronous and asynchronous messaging

WebSphere MQ 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 definition, 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 **mqeQueueManager_triggerTransmission()** function. 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.

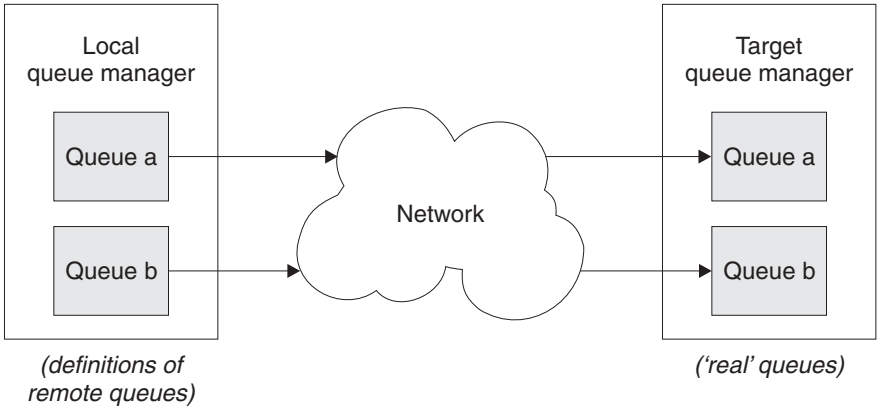

Figure 9. WebSphere MQ Everyplace message flow

Messages are transmitted from the local queue manager to the remote queue 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 WebSphere MQ Everyplace administration messages (see Chapter 6, "Administering messaging resources", on page 55).

The combination of synchronous and asynchronous messaging allows WebSphere MQ Everyplace to cope with unreliable communications links. If a putMessage fails on a synchronous queue, then you have the opportunity to put the message to an asynchronous queue. An example of this is shown below. By defining two queues the application can handle a situation where synchronous transmission is not possible.

Assured message delivery

Asynchronous transmission introduces the concept of *assured message delivery*. When delivering messages asynchronously, WebSphere MQ 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 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 or reuses a connection to the destination queue manager.

The message to be sent is dumped to create a byte-stream, and this byte stream is given to the connection for transmission. Once program control has returned from the connection, 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 WebSphere MQ Everyplace applications.

However, a problem can occur if the sender receives an exception over the connection 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 WebSphere MQ 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 **mqeQueueManager_putMessage** function dictates whether the confirm flow is sent or not. A value of `zero` means that message transmission occurs in one flow, while a value of greater than zero means that a confirm flow is expected. The target queue manager logs the message to the destination queue as usual, but the message is locked and invisible to WebSphere MQ Everyplace applications, until the confirm flow is received.

An WebSphere MQ Everyplace application can issue a **put** message confirmation using the **mqeQueueManager_confirmPutMessage** function. Once the target queue manager receives the flow generated by this command, it unlocks the message, and makes it visible to WebSphere MQ Everyplace applications. You can confirm only one message at a time, it is not possible to confirm a batch of messages.

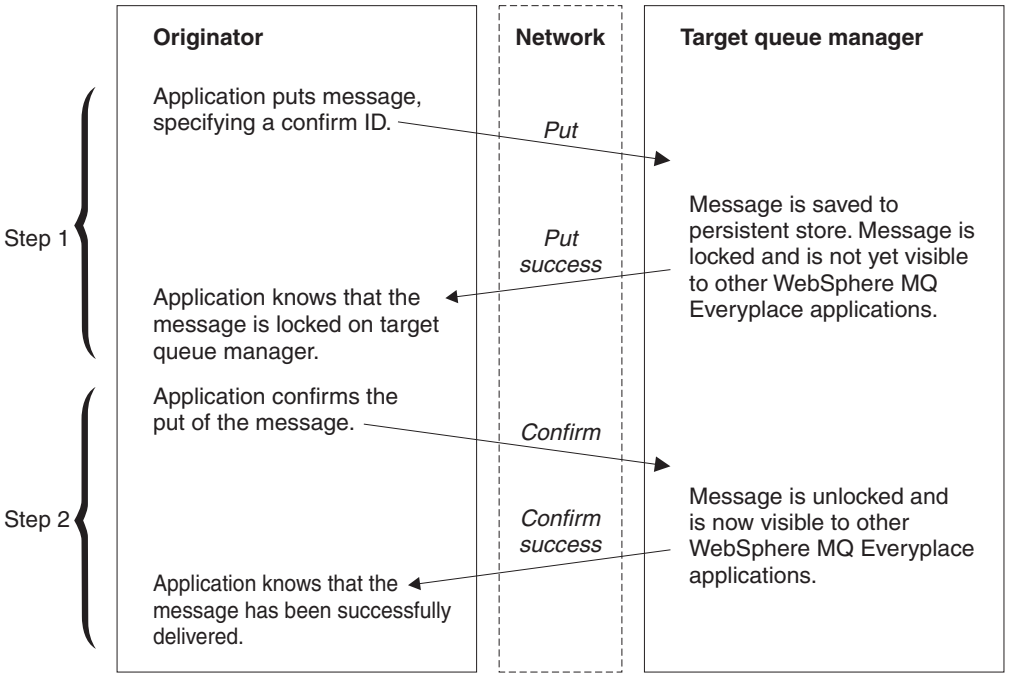

Figure 10. Assured put of synchronous messages

The **mqeQueueManager_confirmPutMessage()** function 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 51.

A skeleton version of the code required for an assured **put** is shown below:

```
MQEINT64 confirmID;
rc = mqe_uniqueValue(pExceptBlock,&confirmID);
if (MQERETURN_OK == rc)
{
 rc = mqeQueueManager_putMessage(hQmgrHandle, pExceptBlock,
             hQueueManagerName,
                                    hQueueName,hOutMsg,NULL,
             confirmID;
 if (MQERETURN_OK == rc)
      {
  MQeFieldsHndl hFilter;
  rc = mqeMsg_getMsgUIDFields(hOutMsg, pExceptBlock,
            &hFilter);
  /* ... ideally check error here */
  rc = mqeQueueManaber_confirmPutMessage(hQMgrHandle, pExceptBlock,
                                          hQueueManagerName,
                hQueueName,
                                         hFilter}
  if (MQERETURN_OK == rc) {
      } else {  /* ... error checking here */}
 } else {/* ... error checking here */ }
} else {  /* ... error checking here */ }
```

If a failure occurs during step 1 in Figure 10 on page 48 the application should retransmit the message. There is no danger of introducing duplicate messages into the WebSphere MQ 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 WebSphere MQ 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 10 on page 48 the application should send the confirm flow again. There is no danger in doing this since the target queue manager ignores any confirm flows it receives for messages that it has already confirmed. This is shown in the following example.

```
   MQEBOOL  msgPut       = MQE_FALSE;
/* put message successful? */
   MQEBOOL  putConfirmed = MQE_FALSE;
/* put confirm successful? */
   MQEINT64 confirmId;
   MQERETURN rc;

   /* generate a confirm Id        */
   rc = mqe_uniqueValue(pErrorBlock, &confirmID);

   if (MQERETURN_OK == rc) {
      MQEINT32 retry;

      /* try to put message - retry if there are problems
    until maximum retry*/
  /* count is exceeded    */
      for (retry=0; (MQE_FALSE == msgPut)
   && (retry < MAX_RETRY); retry++ ) {
         rc = mqeQueueManager_putMessage(hQMgr, pErrorBlock,
               hQueueManager, hQueue, hOutMsg,
               NULL, confirmId );
         if (MQERETURN_OK == rc) {
            /* put successful */
            msgPut = MQE_TRUE;
         } else {
             /* set the message resend flag */
```

```
            /* (error block pointer set to NULL to avoid *pErrorBlock*/
            /*  being overwritten) */
            rc = mqeFields_putBoolean((MQeFieldsHndl)hOutMsg,
                                      NULL, MQE_MSG_RESEND, MQE_TRUE );
            if (MQERETURN_OK != rc) {
               printf("mqeFields_putBoolean error");
               break;
            }
         }
      }

      if (MQE_FALSE == msgPut) {
         /* Number of retries has exceeded the maximum allowed,
    so abort the put */
         /* message attempt */
         /* Attempt to delete any copy of the message held
     on the target queue   */
         /* This operation may well fail */
         /* if the message does exist on the target queue
     then it will remain    */
         /* there until it expires (if an expiry value has
     been set on either    */
         /* the message or queue.  */
         rc = mqeQueueManager_undo(hQMgr, NULL, hQueueManager,
             hQueue, confirmId);
         if (MQERETURN_OK != rc) {
            printf("mqeQueueManager_undo error");
         }
          /* return with put error */
         return pErrorBlock->ec;
      } else {
         MQeFieldsHndl hFilter;

         /* put message unsuccessful */
         /* get message Unique ID for use on confirm put operation */
         rc = mqeMsg_getMsgUIDFields(hOutMsg, pErrorBlock,
             &hFilter);
         if (MQERETURN_OK == rc) {
            /* try to confirm the previous put - retry if
     there are problems until */
            /* maximum retry count is exceeded  */
            for (retry=0; (MQE_FALSE == putConfirmed)
     && (retry < MAX_RETRY); retry++){
               rc = mqeQueueManager_confirmPutMessage(hQMgr, pErrorBlock,
                                                      hQueueManager,
                   hQueue, hFilter );
               if ((MQERETURN_OK == rc) || (MQERETURN_J_NOT_FOUND == rc)) {
                  /* confirm successful */
                  /* An MQERETURN_J_NOT_FOUND exception means
       that the message */
                  /* has already been confirmed */
                  putConfirmed = MQE_TRUE;
               }
               /* another type of exception - need to reconfirm message */
            }

            if ( MQE_FALSE == putConfirmed ) {
               /* Attempt to undo any copy of the message held on
       the target queue */
               /* This operation may well fail  */
               rc = mqeQueueManager_undo(hQMgr, NULL, hQueueManager,
                   hQueue, confirmId);
               if (MQERETURN_OK != rc) {
                  printf("mqeQueueManager_undo error");
               }
               /* return with confirmPut error   */
               return pErrorBlock->ec;
```

```
            }
            (void)mqeFields_free(hFilter, NULL);
         } else {
            displayError("mqeMsg_getMsgUIDFields error", pErrorBlock);
         }
      }
   } else {
      displayError("mqe_uniqueValue error", pErrorBlock);
   }
```

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.

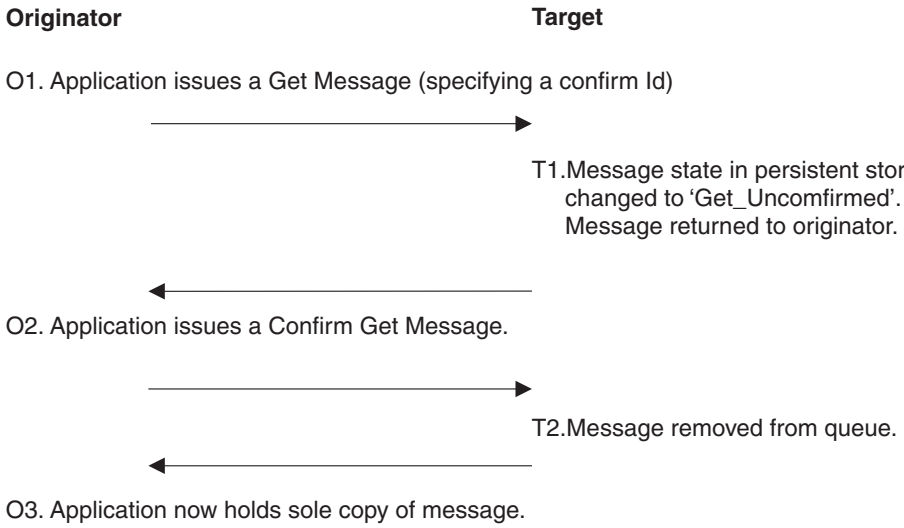

Figure 11. Assured get of synchronous messages

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 WebSphere MQ Everyplace applications.

However, the application that issued the **get** command can restore the messages using the **undo** function. 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.

```
   MQEBOOL  msgGet       = MQE_FALSE;
 /* get message successful?    */
   MQEBOOL  getConfirmed = MQE_FALSE;
/* get confirm successful?    */
   MQEINT64 confirmId;
   MQERETURN rc;

   *phInMsg = NULL;

   /* generate a confirm Id */
   rc = mqe_uniqueValue(pErrorBlock,&confirmID);
```

```
if (MQERETURN_OK == rc) {
   MQEINT32 retry;

   /* try to put message - retry if there are
problems until maximum retry     */
   /* count is exceeded         */
   for (retry=0; (MQE_FALSE == msgGet) &&
    (retry < MAX_RETRY); retry++ ) {
      rc = mqeQueueManager_getMessage(hQMgr, pErrorBlock, phInMsg,
            hQueueManager,
                                      hQueue, hFilter, NULL,
                                      confirmId);
      if (MQERETURN_OK == rc) {
         /* put successful */
         msgGet = MQE_TRUE;
      } else if (MQERETURN_J_Q_NO_MATCHING_MSG == rc) {
         /* the message is unavailable */
         break;
      } else {
         /* 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  */
         rc = mqeQueueManager_undo(hQMgr, NULL, hQueueManager,
                                   hQueue, confirmId);
         if (MQERETURN_OK != rc) {
            printf("mqeQueueManager_undo error");
         }
      }
   }

   if (MQE_FALSE == msgGet) {
      /* Number of retry attempts has exceeded
  the maximum allowed, so abort    */
      /* get message operation */
      return pErrorBlock->ec;
   } else {
      /* try to confirm the previous get
  message operation - retry if there are */
      /* problems until maximum retry count is exceeded */
      for (retry=0; (MQE_FALSE == getConfirmed) &&
  (retry < MAX_RETRY); retry++) {
         rc = mqeQueueManager_confirmGetMessage(hQMgr, pErrorBlock,
               hQueueManager,
                                                hQueue, hFilter);
         if (MQERETURN_OK == rc) {
            getConfirmed = MQE_TRUE;
         }
      }

      if ( MQE_FALSE == getConfirmed ) {
         /* need to free the message already got and return error */
         (void)mqeMsg_free(*phInMsg, NULL);
         return pErrorBlock->ec;
      }
   }
} else {
   displayError("mqe_uniqueValue error", pErrorBlock);
}
```

The **undo** command also has relevance for the **mqeQueueManager_putMessage** and **mqeQueueManager_browseMessagesAndLock** commands. As with **get** message, the **undo** command restores any messages locked by the **mqeQueueManager_browseMessagesandLock** command to their previous state.

If an application issues an **undo** command after a failed **mqeQueueManager_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.

Security

The queue manager fully supports the security functions supplied with WebSphere MQ Everyplace. Any messages stored in a queue defined with security characteristics are encoded using those characteristics. Any communication connections set up between a queue manager and a secure queue use the security characteristics of the queue, or an existing connections with equal or higher 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 111 for a detailed discussion of WebSphere MQ Everyplace security.

Chapter 6. Administering messaging resources

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

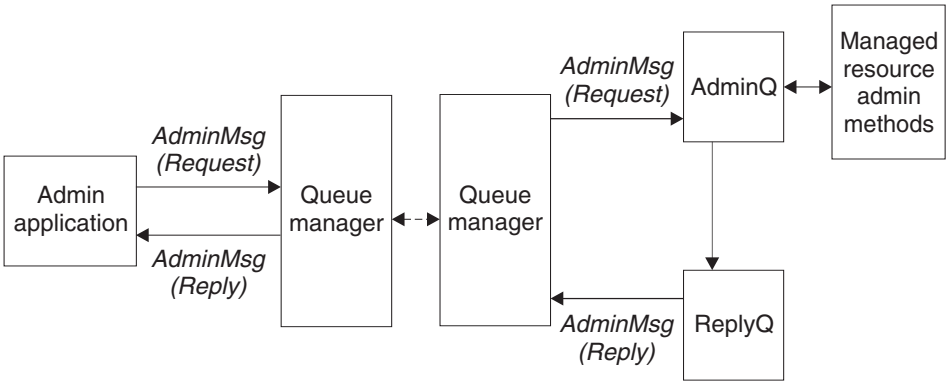

Figure 12. WebSphere MQ Everyplace administration

Before you can administer a queue manager or its resources, you must start the queue manager and configure an administration queue 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 **mqeQueueManagerConfigure_defineDefaultAdminQueue()** function 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 **mqeQueueManagerConfigure_defineDefaultAdminReplyQueue()** function 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 WebSphere MQ Everyplace resources:

Table 2. Administration messages

Message name	purpose
MQe_AdminMsg	an abstract class that acts as the base class for all administration messages
MQe_AdminQueueAdminMsg	provides support for administering the administration queue
MQe_ConnectionAdminMsg	provides support for administering connections between queue managers
MQe_HomeServerQueueAdminMsg	provides support for administering home-server queues
MQe_QueueAdminMsg	provides support for administering local queues
MQe_QueueMangerAdminMsg	provides support for administering queue managers
MQe_RemoteQueueAdminMsg	provides support for administering remote queues
MQe_StoreAndForwardQueueAdminMsg	provides support for administering store-and-forward queues

The basic administration request message

Every request to administer an WebSphere MQ Everyplace resource takes the same basic form. The following list describes 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.
   - MQE_ADMIN_TARGET_QMGR
   - MQE_ADMIN_ACTION
   - MQE_ADMIN_MAXATTEMPTS
2. Administration fields, that are specific to the resource being managed.
   - MQE_ADMIN_PARAMS (charactistics of managed resources required for the action)
   - MQE_ADMIN_NAME
   - Other related fields
3. Optional fields to assist with the processing of administration messages.
   - MQE_MSG_STYLE
   - MQE_MSG_REPLYTO_Q
   - MQE_MSG_REPLYTO_QMGR
   - MQE_MSG_MSGID
   - MQE_MSG_CORRELID

Base administration fields

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

MQE_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.

MQE_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 3. Administration actions

Administration action	Purpose
MQE_ADMIN_ACTION_CREATE	Create a new instance of a managed resource.
MQE_ADMIN_ACTION_DELETE	Delete an existing managed resource
MQE_ADMIN_ACTION_INQUIRE	Inquire on one or more characteristics of a managed resource
MQE_ADMIN_ACTION_INQUIREALL	Inquire on all characteristics of a managed resource
MQE_ADMIN_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 function that sets the *MQE_ADMIN_ACTION* field:

Table 4. Setting the administration action field

Administration action	Setting function
MQE_ADMIN_ACTION_CREATE	mqeAdminMsg_create(*parameters*)
MQE_ADMIN_ACTION_DELETE	mqeAdminMsg_delete(*parameters*)
MQE_ADMIN_ACTION_INQUIRE	mqeAdminMsg_inquire(*parameters*)
MQE_ADMIN_ACTION_INQUIREALL	mqeAdminMsg_inquireAll(*parameters*)
MQE_ADMIN_ACTION_UPDATE	mqeAdminMsg_update(*parameters*)

Where *parameters* is:

```
MQeAdminMsgHndl hMsgObj,
MQeExceptBlock *pErrStruct,
MQECONST MQeFieldsHndl hParms
```

MQE_ADMIN_MAXATTEMPTSAdmin_MaxAttempts

This field determines how many 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. Refer to "The basic administration reply message" on page 61 for more details on using the reply data.

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

Table 5. Setting administration request fields

Administration action	Field type	Set and get functions
MQE_ADMIN_PARAMS	MQeFields	mqeAdminMsg_getInputFields(MQeAdminMsgHndl hMsgObj , MQeExceptBlock * pErrStruct , MQeFieldsHndl * phFields)
MQE_ADMIN_ACTION	int	mqeAdminMsg_setAction(MQeAdminMsgHndl hMsgObj, MQeExceptBlock * pErrStruct, MQEINT32 action)
MQE_ADMIN_TARGET_QMGR	ascii	mqeAdminMsg_setTargetQMgr(MQeAdminMsgHndl hMsgObj, MQeExceptBlock * pErrStruct, MQECONST MQeStringHndl hName)
MQE_ADMIN_MAXATTEMPTS	int	mqeAdminMsg_setMaxAttempts(MQeAdminMsgHndl hMsgObj, MQeExceptBlock * pErrStruct, MQEINT32 maxAttempts)

Fields specific to the managed resource

MQE_ADMIN_PARAMS

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 *MQE_ADMIN_NAME*, and it has a type of ascii.

The full set of characteristics of a resource can be determined by using the **mqeAdminMsg_characteristics()** function against an instance of an administration message. This function returns an MQeFields object that contains one field for each characteristic. MQeFields functions 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, *MQE_ADMIN_NAME*, must be passed. In the case of **MQE_ADMIN_ACTION_INQUIRE** this is the only parameter that is required.

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.

MQE_MSG_STYLE

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

MQE_MSG_STYLE_DATAGRAM
A command not requiring a reply

MQE_MSG_STYLE_REQUEST
A request that would like a reply

MQE_MSG_STYLE_REPLY
A reply to a request

If *MQE_MSG_STYLE* is set to *MQE_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:

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

MQE_MSG_REPLYTO_QMGR
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. WebSphere MQ Everyplace provides two fields that can be used for this purpose:

MQE_MSG_MSGID
A byte array containing the message ID

MQE_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:

```
MQERETURN primeAdminMsg(MQeQueueAdminMsgHndl hMsg, MQeExceptBlock
         *pErrorBlock, MQeFieldsHndl * phMsgTest,
         MQeStringHndl hQueueManagerName) {

   MQERETURN rc;
   /* Set the target queue manager that will process this message */
   rc = mqeAdminMsg_setTargetQMgr((MQeAdminMsgHndl)hMsg,
                                  pErrorBlock, hQueueManagerName);
   if (MQERETURN_OK == rc) {
      /* Ask for a reply message */
      rc = mqeFields_putInt32((MQeFieldsHndl)hMsg, pErrorBlock,
                              MQE_MSG_STYLE, MQE_MSG_STYLE_REQUEST);
      if (MQERETURN_OK == rc) {
         rc = mqeFields_putAscii((MQeFieldsHndl)hMsg, pErrorBlock,
                                 MQE_MSG_REPLYTO_Q, MQE_ADMIN_REPLY_QUEUE_NAME);
         if (MQERETURN_OK == rc) {
            rc = mqeFields_putAscii((MQeFieldsHndl)hMsg, pErrorBlock,
                                    MQE_MSG_REPLYTO_QMGR, hQueueManagerName);
            if (MQERETURN_OK == rc) {
               rc = mqeFields_new(pErrorBlock, phMsgTest);
               if (MQERETURN_OK == rc) {
                  MQEINT64 v;
                  MQeStringHndl hFieldName;

                  /* create some identical data fields in both hMsg and *phMsgTest
```

```
                        so that the replay message can be matched against when it is
                        returned to the replyToQ.
                      */
                     rc = mqe_uniqueValue(pErrorBlock, &v);
                     if (MQERETURN_OK == rc) {
                        rc = mqeString_newChar8(pErrorBlock, &hFieldName,
                   "IDField" );
                        if (MQERETURN_OK == rc) {
                           rc = mqeFields_putInt64(*phMsgTest, pErrorBlock,
                    hFieldName, v);
                           if (MQERETURN_OK == rc) {
                              rc = mqeFields_putInt64((MQeFieldsHndl)hMsg,
                                                      pErrorBlock, hFieldName, v);
                              if (MQERETURN_OK != rc) {
                                 displayError("mqeFields_putInt64 error
                                              (in primeAdminMsg, hMsg)", pErrorBlock);
                              }
                           } else {
                              displayError("mqeFields_putInt64 error (in primeAdminMsg,
                                           phMsgTest)", pErrorBlock);
                           }
                           (void)mqeString_free(hFieldName, NULL);
                        } else {
                           displayError("mqeString_newChar8 error (in primeAdminMsg)",
                                        pErrorBlock);
                        }
                     } else {
                        displayError("mqe_uniqueValue error (in primeAdminMsg)",
                                     pErrorBlock);
                     }
                  } else {
                     displayError("mqeFields_new error (in primeAdminMsg)",
                                  pErrorBlock);
                  }
               } else {
                  displayError("mqeFields_putAscii (2) error (in primeAdminMsg)",
                               pErrorBlock);
               }
            } else {
               displayError("mqeFields_putAscii (1) error (in primeAdminMsg)",
                            pErrorBlock);
            }
         } else {
            displayError("mqeAdminMsg_Int32 error (in primeAdminMsg)",
                         pErrorBlock);
         }
      } else {
         displayError("mqeAdminMsg_setTargetQMgr error (in primeAdminMsg)",
                      pErrorBlock);
      }

      return rc;
   }
```

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

Standard WebSphere MQ 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. Administration example Ex1 contains code that demonstrates these functions.

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.

A reply is made up of:

1. Base administration fields. These are copied from the request message.
   - MQE_ADMIN_TARGET_QMGR
   - MQE_ADMIN_ACTION
   - MQE_ADMIN_MAXATTEMPTS
2. Administration fields that are specific to the resource being managed.
   - MQE_ADMIN_PARAMS (characteristics of managed resource required for the action)
   - MQE_ADMIN_NAME
   - Others
3. Optional fields to assist with the processing of administration messages. These are copied from the request message.
   - MQE_MSG_STYLE
   - MQE_MSG_REPLYTO_Q
   - MQE_MSG_REPLYTO_QMGR
   - MQE_MSG_MSGID
   - MQE_MSG_CORRELID
4. Administration fields detailing outcome of request.
   - MQE_ADMIN_RC
   - MQE_ADMIN_REASON
   -
5. Administration fields providing detailed results of the request that are specific to the resource being managed.
   - MQE_ADMIN_PARAMS (characteristics of managed resource required for the action)
   - MQE_ADMIN_NAME
   - Other related fields
6. Administration fields detailing errors that are specific to the resource being managed.
   - MQE_ADMIN_ERROR (error field items, one per characteristic in error)
   - Field in error
   - Other related errors

The first three items are describe in “The basic administration request message” on page 56. The reply specific fields are described in the following sections.

Outcome of request fields

MQE_ADMIN_RC_FIRLD

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

MQE_ADMIN_RC_SUCCESS
The action completed successfully.

MQE_ADMIN_RC_FAIL
The request failed completely.

MQE_ADMIN_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.

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

MQE_ADMIN_PARAMS
An MQeFields object containing a field for each characteristics of the managed resource.

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

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

Table 6. Getting administration reply fields

Administration field	Field type	get function
MQE_ADMIN_RC	int	mqeAdminMsg_getAction(MQeAdminMsgHndl hMsgObj , MQeExceptBlock * pErrStruct , MQEINT32 * pAction)
MQE_ADMIN_REASON	unicode	mqeAdminMsg_getReason(MQeAdminMsgHndl hMsgObj , MQeExceptBlock * pErrStruct , MQeStringHndl * phReason)
MQE_ADMIN_PARAMS	MQeFields	mqeAdminMsg_getOutputFields(MQeAdminMsgHndl hMsgObj , MQeExceptBlock * pErrStruct, MQeFieldsHndl * phFields)
MQE_ADMIN_ERROR	MQeFields	mqeAdminMsg_getErrorFields(MQeAdminMsgHndl hMsgObj , MQeExceptBlock * pErrStruct , MQeFieldsHndl * phErrors)

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 *MQE_QUEUE_DESCRIPTION* and *MQE_QUEUE_FILEDESC*, the resultant MQeFields object would contain the values for the actual queue in these two fields.

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

Table 7. Enquiring on queue parameters

Admin_Parms field name	Request message		Reply message	
	Type	Value	Type	Value
MQE_ADMIN_NAME	ascii	″TestQ″	ascii	″TestQ″
MQE_QUEUE_MANAGER_NAME	ascii	″ExampleQM″	ascii	″ExampleQM″

Table 7. Enquiring on queue parameters (continued)

Admin_Parms field name	Request message		Reply message	
	Type	Value	Type	Value
MQE_QUEUE_DESCRIPTION	Unicode	null	Unicode	″A test queue″
MQE_QUEUE_FILEDESC	ascii	null	ascii	″c:\queues\″

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

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 `MQE_ADMIN_PARAMS` field for a request to update a queue and the resultant `MQE_ADMIN_ERROR` field:

Table 8. Request and reply message to update a queue

Field name	Request message		Reply message	
	Type	Value	Type	Value
MQE_ADMIN_PARAMS field				
MQE_ADMIN_NAME	ascii	″TestQ″	ascii	″TestQ″
MQE_QUEUE_MANAGER_NAME	ascii	″ExampleQM″	ascii	″ExampleQM″
MQE_QUEUE_DESCRIPTION	Uni-code	null	Uni-code	″ExampleQM″ ″A new description″
MQE_QUEUE_FILEDESC	ascii	null	Uni-code	″D:\queues″
MQE_ADMIN_ERROR field				
MQE_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 example program Admin-Ex1 contains code showing how to look at the error string. (In the **showReasons()** function.)

Administration of managed resources

As described in previous sections, WebSphere MQ 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 *WebSphere MQ Everyplace 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 27 for information on creating and starting a queue manager.

The queue manager has very few characteristics itself, but it controls other WebSphere MQ 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 MQeConnectionAdminMessage to 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:

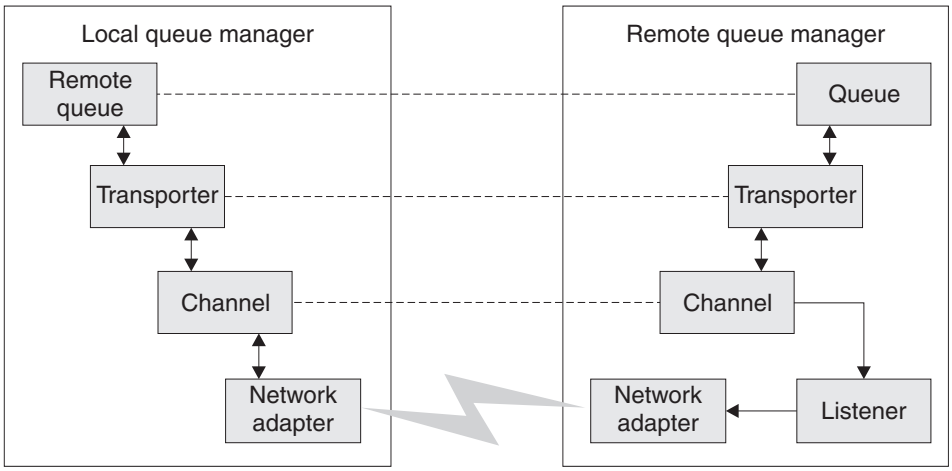

Figure 13. Queue manager connections

Communication happens at different levels:

Transporter:
Logical connection between two queues

Connection:
Logical connection between two systems

Adapter:
Protocol specific communication

The connection and adapter are specified as part of a connection definition. The transporter is specified as part of a remote queue definition.

To create a suitable Connection Admin Msg, use the **primeAdminMsg** function shown in the Example files Admin-Ex1. Set the Remote Queue Manager name,

using the **mqeAdminMsg_setName** function, and then call the **mqeConnectionAdminMsg_create** function.

To add a connection :

1. Create a new MQeConnectionAdminMsg
2. Prime the admin message
3. Set the remote queuemanager name
4. Call **mqeConnectionAdminMsg_create**

WebSphere MQ Everyplace provides a choice of connection 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 to server configuration, one queue manager acts as a client and the other runs in a server environment. A server allows multiple simultaneous incoming connections. To accomplish this the server must have components that can handle multiple incoming requests. See "Server queue managers" on page 36 for a description of how to run a queue manager in a server environment.

Figure 14 shows the typical connection components in a client to server configuration.

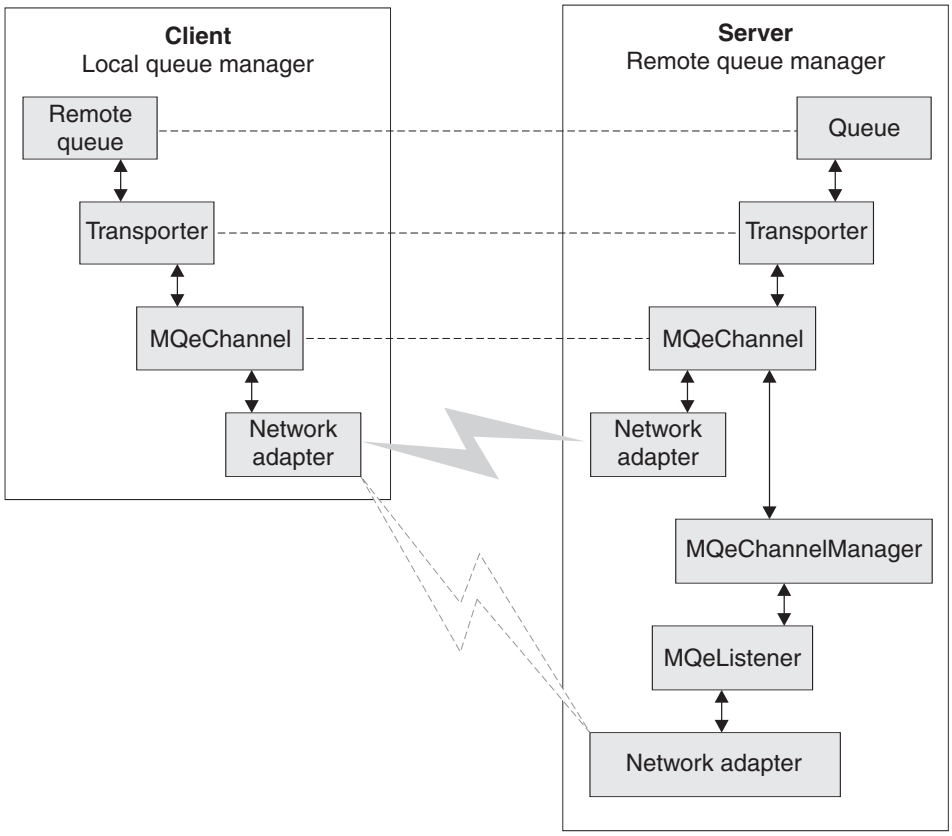

Figure 14. Client to server connections

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

Create a connection, as described previously and then use the standard putMessage calls to send the message to the administration queue.

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 functions 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 "Class Aliases" on page 33). When an application calls functions 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 **MQE_ADMIN_ACTION_ADDALIAS** and **MQE_ADMIN_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 functions **addAlias()** or **removeAlias()**. Each of these functions takes one alias name but you can call the function repeatedly to add multiple aliases to a message.

Queues

The queue types provided by WebSphere MQ Everyplace are described briefly in "WebSphere MQ 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 WebSphere MQ 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

MQE_ADMIN_NAME and *MQE_QUEUE_QMGRNAME*. You can use the **setName(queueManagerName, queueName)** function to set these two fields in the administration message.

For example, if you configure a queue manager with a local queue, and queue manager is called `qm1` and the local queue is called `invQ`. The queue manager name characteristic of the queue is qm1, which matches the queue manager name, that is

```
putMessage(null, invQ, msg, ...)
msg = getMessage(null, invQ, ...)
```

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
```

WebSphere MQ Everyplace Version 2.0.0.5 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 database 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 5, “Queue managers, messages, and queues”, on page 27 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.

Creating a local queue: To create a local queue:

1. Create a new MQeQueueAdminMsg
2. Prime the administration message
3. Call **mqeQueueAdminMsg_setName()** function to set the queue and queue manager name

4. Create an MQeFields object containing entries for the characteristics. Typical entries would be:
   - MQE_QUEUE_DESCRIPTION for the descriptions
   - MQE_QUEUE_FILEDESC for the queue store locations
5. Call the mqeAdminMsg_create function
6. Put the message to the administration queue

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:

- *MQE_QUEUE_CRYPTOR*
- *MQE_QUEUE_AUTHENTICATOR*
- *MQE_QUEUE_COMPRESSOR*
- *MQE_QUEUE_TARGETREGISTRY*
- *MQE_QUEUE_ATTRULE*

For more detailed information on setting up queue based security see Chapter 8, “Security”, on page 111.

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 *WebSphere MQ Everyplace Programming Reference*.

Aliases: Queue names can have aliases similar to those described for connections in “Aliases” on page 66. 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 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 56 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 *MQE_ADMIN_NAME* and *MQE_QUEUE_QMGRNAME*. You can use the **setName(queueManagerName, queueName)** function 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, WebSphere MQ 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 *MQE_QUEUE_MODE* field to MQE_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 67).

To set a queue for asynchronous operation, set the *MQE_QUEUE_MODE* field to MQE_QUEUE_ASYNCHRONOUS.

Figure 15 on page 70 shows an example of a remote queue set up for synchronous operation and a remote queue setup for asynchronous operation.

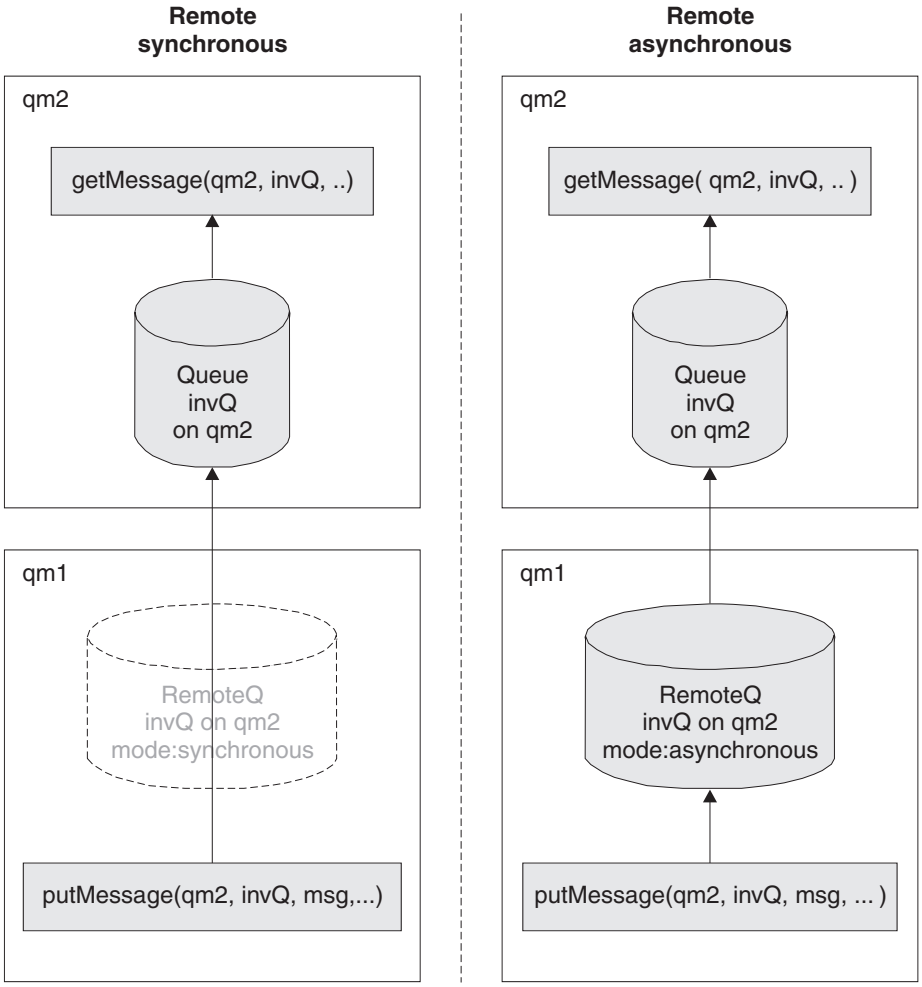

Figure 15. Remote queue

- In both the synchronous and asynchronous examples queue manager `qm2` has a local queue `invQ`
- 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: To create a remote queue:

1. Create a new MQeRemoteQueueAdminMsg

2. Prime this administration message
3. Set the queue and queue manager names
4. Create an MQeFields object to hold the characteristics (Typically, description, queue mode, and queue store)
5. Call the **create** function
6. Put the message to administration queue

For synchronous operation, the queue characteristics for inclusion in the remote queue definition can be obtained using *queue discovery* which is explained on page 46.

Store-and-forward queue

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

- Forward messages either to the target queue manager, or to another queue manager between the sending and the target queue managers. In this case the store-and-forward queue pushes messages either to the next hop or to the target queue manager
- 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 73). 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 **MQE_ACTION_ADD_QMGR** and **MQE_ACTION_REMOVE_QMGR** 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 `MQE_QUEUE_QMGRNAMELIST`. Alternatively you can use the **addQueueManager()** and **removeQueueManager()** functions. Each of these functions takes one queue manager name but you can call the function repeatedly to add multiple queue managers to a message.

To add a queue manager:

1. Create a new MQeStoreAndForwardQueueAdminMsg object
2. Prime the administration message
3. Set queue and queue manager names
4. Call the **mqeStoreAndForwardQueueAdminMsg_addQueueManager** function

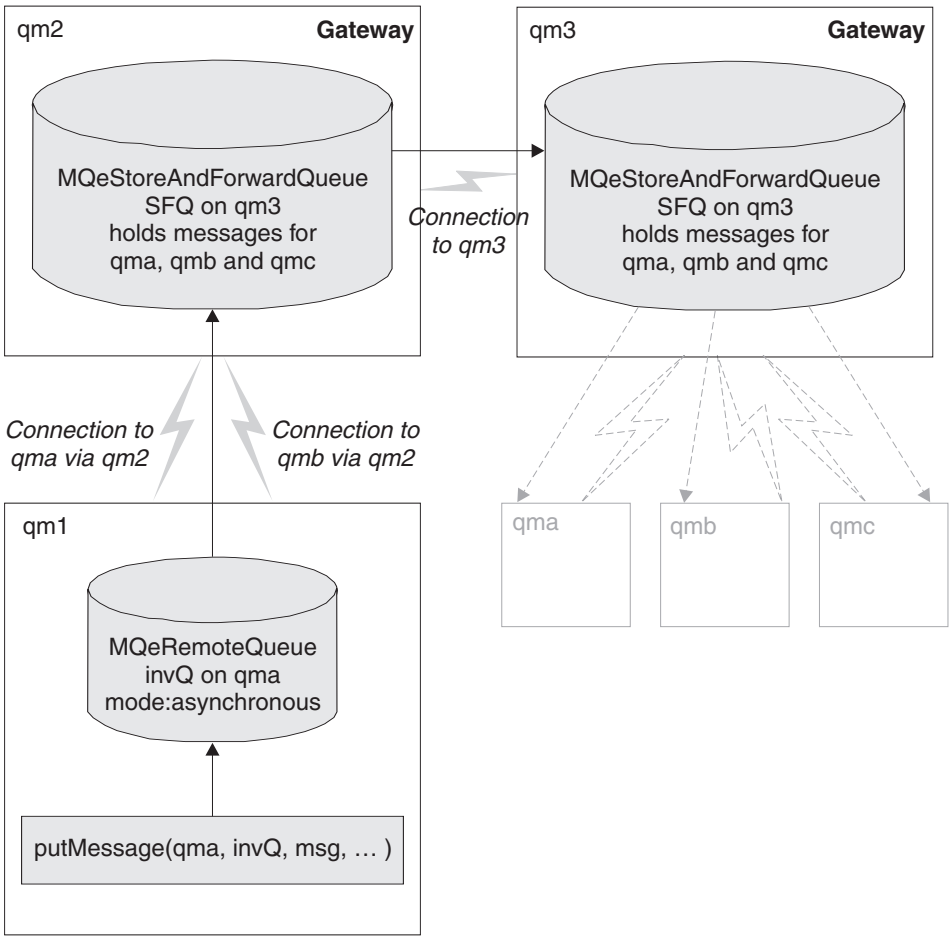

Figure 16. Store-and-forward queue

Each store-and-forward queue has to be configured to handle messages for any queue managers for which it can hold messages. Use the **MQE_ACTION_ADD_QMGR** action, described earlier in this section, to add the queue manager information to each 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. A connection with the same name as the next queue manager must also be configured. The store-and-forward queue uses this connection as the transport mechanism for pushing messages to the next hop.

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 (but it must still be configured). 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 16 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 temporarily holds messages on behalf of qma, qmb and qmc, (but not qm3).
- 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 16 on page 72 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 *MQE_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. A value of `0` for this field means that the home-server is only polled when the **mqeQueueManager_triggerTransmission** function is called

Note: If a home-server queue fails to connect to its store-and-forward queue (for instance if the store-and-forward queue is unavailable when the home server queue starts) it will stop trying until a trigger transmit call is made.

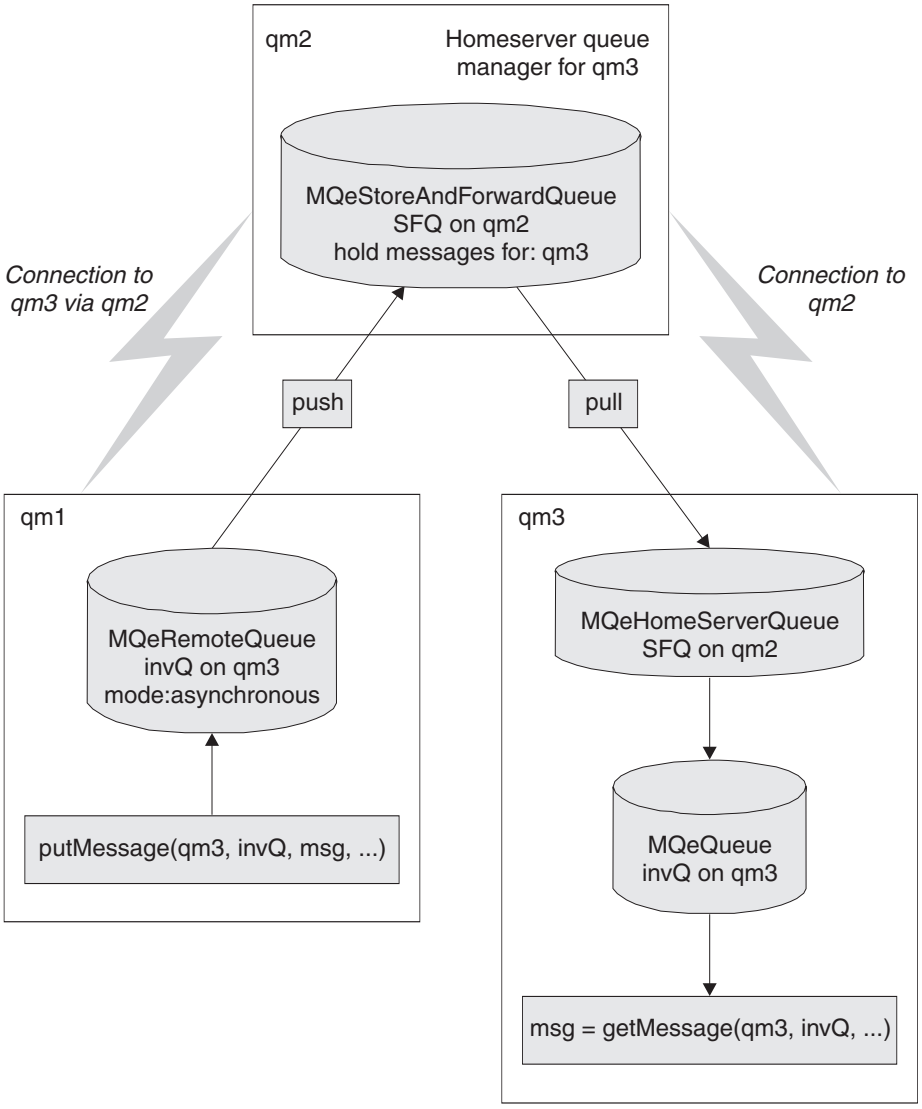

Figure 17. Home-server queue

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 17 on page 74 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.

WebSphere MQ bridge queue

A WebSphere MQ bridge queue is a remote queue definition that refers to a queue residing on a WebSphere MQ queue manager. The queue holding the messages resides on the WebSphere MQ queue manager, not on the local queue manager.

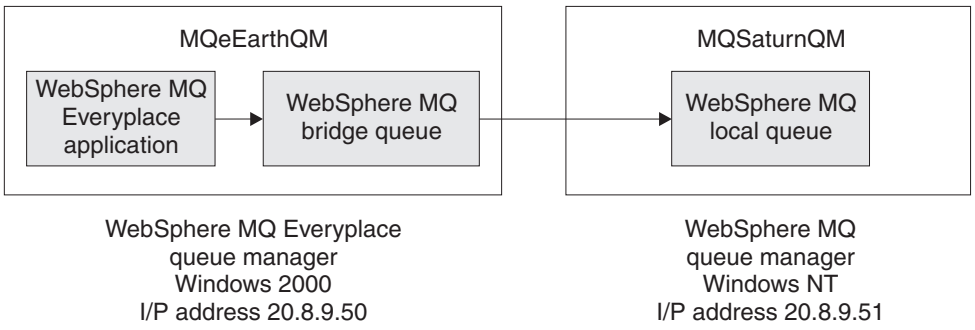

Figure 18. WebSphere MQ bridge queue

- The MQSaturnQM WebSphere MQ queue manager has a local queue MQSaturnQ defined .
- The MQeEarthQM must have an WebSphere MQ bridge queue defined called MQSaturnQ on the MQSaturnQM queue manager.
- Applications attached to the MQeEarthQM queue manager put messages to the MQSaturnQ WebSphere MQ 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, WebSphere MQ queue manager proxy, and client connection names are specified to uniquely identify a client connection object in the bridge object hierarchy (see Figure 19 on page 80). This information identifies how the WebSphere MQ bridge accesses the WebSphere MQ queue manager, to manipulate a WebSphere MQ queue.

The WebSphere MQ bridge queue provides the facility to put to a queue on a queue manager that is not directly connected to the WebSphere MQ bridge. This

allows a message to be sent to a WebSphere MQ queue manager (the target) routed through another WebSphere MQ queue manager. The WebSphere MQ bridge queue takes the name of the target queue manager and the intermediate queue manager is named by the WebSphere MQ queue manager proxy.

For a complete list of the characteristics used by the WebSphere MQ bridge queue, see the WebSphere MQ bridge section of the *WebSphere MQ Everyplace C Programming Reference*..

Table 9 details the list of operations supported by the WebSphere MQ bridge queue, once it has been configured:

Table 9. Message operations supported by WebSphere MQ—bridge queue

Type of operation	Supported by WebSphere MQ bridge queue
getMessage()	yes*
putMessage()	yes
browseMessage()	Yes*
browseAndLockMessage	no
Note: * These functions have restrictions on their use. See "Getting and browsing messages from the WebSphere MQ bridge queue" on page 106	

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 WebSphere MQ queue manager from the pool of connections maintained by the bridge's client connection object. The logical connection to WebSphere MQ is supplied by either the WebSphere MQ Java Bindings classes, or the WebSphere MQ Classes for Java. The choice of classes depends on the value of the *hostname* field in the WebSphere MQ queue manager proxy settings. Once the WebSphere MQ bridge queue has a connection to the WebSphere MQ queue manager, it attempts to put the message to the WebSphere MQ queue.

An WebSphere MQ 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 WebSphere MQ bridge queue and returns success, your message has passed to the WebSphere MQ system while your process was waiting for the put operation to complete.

If you do not wish to use synchronous operations against the WebSphere MQ bridge queue, you may set up an asynchronous remote queue definition (see "Asynchronous messaging" on page 45) that refers to the WebSphere MQ 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()** function returns, the message may not necessarily have passed to the WebSphere MQ queue manager.

An example of WebSphere MQ bridge queue usage is described in "Configuration example" on page 83.

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 56 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 with an MQeException. 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 *MQE_QUEUE_QTIMERINTERVAL* in the administration message. The interval value is specified in milliseconds.

Security and administration

By default, any WebSphere MQ 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. WebSphere MQ Everyplace provides the basic facilities for securing administration using queue-based security which is described in Chapter 8, "Security", on page 111.

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.

Chapter 7. WebSphere MQ bridge

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

Installation

Make sure that the following configuration is complete before trying to use an WebSphere MQ bridge.

WebSphere MQ Classes for Java

To use the WebSphere MQ bridge you must have the WebSphere MQ Classes for Java (version 5.1 or greater) installed on your WebSphere MQ Everyplace system. WebSphere MQ Classes for Java is available for free download from the Web as supportpac MA88. The Web address for the download is: `http://www.ibm.com/software/mqseries/txppacs/ma88.html`. (The WebSphere MQ classes for Java for NT is shipped with WebSphere MQ Version 5.1 for NT.)

These classes must be added to the classpath set in the options file - see "JVM options" on page 10 for details on how to specify the classpath.

Configuring the WebSphere MQ bridge

The configuration of the WebSphere MQ bridge requires you to perform some actions on the WebSphere MQ queue manager, and some on the WebSphere MQ Everyplace queue manager. The bridge is logically broken into two pieces, one for each direction of the message (WebSphere MQ Everyplace to WebSphere MQ and WebSphere MQ to WebSphere MQ Everyplace)

The bridge objects are defined in a hierarchy as shown in Figure 19 on page 80

The following rules govern the relationship between the various objects:

- An WebSphere MQ Everyplace bridges object is associated with a single WebSphere MQ Everyplace queue manager.
- A single WebSphere MQ Everyplace bridges object may have more than one bridge object associated with it. You may wish to configure several WebSphere MQ bridge instances with different routings.
- Each bridge can have a number of WebSphere MQ queue manager proxy definitions.
- Each WebSphere MQ queue manager proxy definition can have a number of client connections that allow communication with WebSphere MQ Everyplace.
- Each client connection connects to a single WebSphere MQ queue manager. Each connection may use a different *server connection* on the WebSphere MQ queue manager , or a different set of security, send, and receive exits, ports or other parameters.
- A WebSphere MQ bridge client connection may have a number of transmission queue listeners that use that bridge service to connect to the WebSphere MQ queue manager.

- A listener uses only one client connection to establish its connection.
- Each listener connects to a single transmission queue on the WebSphere MQ system.
- Each listener moves messages from a single WebSphere MQ transmission queue to anywhere on the WebSphere MQ Everyplace network, (through the WebSphere MQ Everyplace queue manager its bridge is associated with). So a WebSphere MQ bridge can funnel multiple WebSphere MQ message sources through one WebSphere MQ Everyplace queue manager onto the WebSphere MQ Everyplace network.
- When moving WebSphere MQ Everyplace messages to the WebSphere MQ network, the WebSphere MQ Everyplace queue manager creates a number of *adapter* objects. Each adapter object can connect to any WebSphere MQ queue manager (providing it is configured) and can send its messages to any queue. So an WebSphere MQ bridge can dispatch WebSphere MQ Everyplace messages routed through a single WebSphere MQ Everyplace queue manager to any WebSphere MQ queue manager.

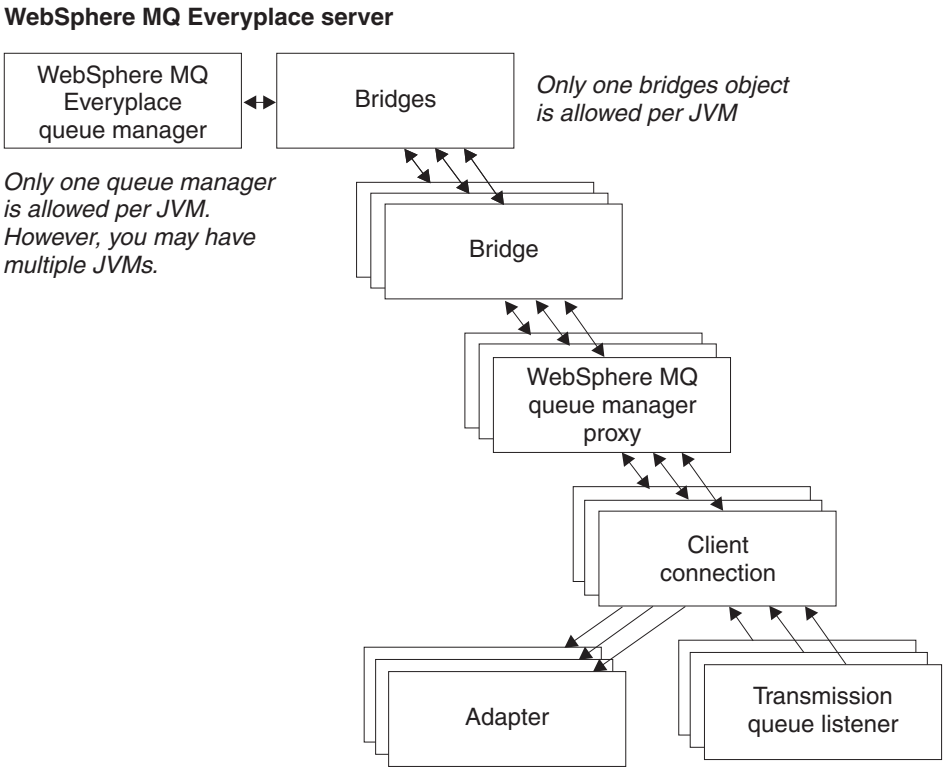

Figure 19. Bridge object hierarchy

Naming recommendations for inter-operability with a WebSphere MQ network

To create an WebSphere MQ Everyplace network that can interoperates with a WebSphere MQ network, it is necessary to adopt the same limitations in naming convention for both systems. It is therefore important to understand the differences between valid queue names in both systems:

- In WebSphere MQ, the forward-slash '/' character is allowed in queue and queue manager names. This character is not valid in WebSphere MQ Everyplace object names.

We strongly recommend that you do not use this character in the name of any WebSphere MQ queue or queue manager.

- WebSphere MQ queue and queue manager names have a limit of 48 characters but WebSphere MQ Everyplace names have no length restrictions.

  We strongly recommend that you do not define WebSphere MQ Everyplace queues or queue managers with names that contain more than 48 characters.
- WebSphere MQ queue names can have leading or trailing '.' characters. This is not allowed in WebSphere MQ Everyplace

  We strongly recommend that you do not defined any WebSphere MQ queue or queue manager with a name that starts or ends with a '.' character.
- Queue managers should be named uniquely, such that a queue manager with the same name does not exist on either the WebSphere MQ Everyplace network, or the WebSphere MQ network.

If you choose not to follow these guidelines, then you may experience problems when trying to address an WebSphere MQ Everyplace queue from a WebSphere MQ application.

Configuring a basic installation

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

1. Make sure you have a WebSphere MQ system installed and that you understand local routing conventions, and how to configure the system.
2. Install WebSphere MQ Everyplace on a system (It can be the same system as your WebSphere MQ system is located on if you wish)
3. If WebSphere MQ Classes for Java is not already installed, download it from the Web and install it. (See "WebSphere MQ Classes for Java" on page 79.)
4. Set up your WebSphere MQ Everyplace system and verify that it is working properly before you try to connect it to WebSphere MQ.
5. Make sure that the supportpac MA88 .jar files are in the classpath. This is set by the *MQE_VM_OPTIONS_LOCN* which should be set to point to the vm_options.txt file during installation. See "Installation" on page 9.
6. Plan the routing you intend to implement. You need to decide which WebSphere MQ queue managers are going to talk to which WebSphere MQ Everyplace queue managers.
7. Decide on a naming convention for WebSphere MQ Everyplace objects and WebSphere MQ objects and document it for future use.
8. Modify your WebSphere MQ Everyplace system to define a WebSphere MQ bridge on your chosen WebSphere MQ Everyplace server.
9. Connect the chosen WebSphere MQ queue manager to the bridge on the WebSphere MQ Everyplace server as follows:
   - On the WebSphere MQ queue manager:

     Define one or more Java server connections so that WebSphere MQ Everyplace can use the WebSphere MQ Classes for Java to talk to this queue manager. This involves the following steps:

     a. Define the server connections
     b. Define a sync queue for WebSphere MQ Everyplace to use to provide assured delivery to the WebSphere MQ system. You need one of these for each server connection that the WebSphere MQ Everyplace system can use.
   - On the WebSphere MQ Everyplace server:

a. Define a WebSphere MQ queue manager proxy object which holds information about the WebSphere MQ queue manager. This involves the following steps:
   1) Collect the Hostname of the WebSphere MQ queue manager.
   2) Put the name in the WebSphere MQ queue manager proxy object.
b. Define a Client Connection object that holds information about how to use the WebSphere MQ Classes for Java to connect to the server connection on the WebSphere MQ system. This involves the following steps:
   1) Collect the Port number, and all other server connection parameters.
   2) Use these values to define the client connection object so that they match the definition on the WebSphere MQ queue manager.

10. Modify your configuration on both WebSphere MQ Everyplace and WebSphere MQ to allow messages to pass from WebSphere MQ to WebSphere MQ Everyplace.
    a. Decide on the number of routes from WebSphere MQ to your WebSphere MQ 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 as follows:
       1) For each route define a transmission queue on your WebSphere MQ system. DO NOT define a connection for these transmission queues.
       2) For each route create a matching transmission queue listener on your WebSphere MQ Everyplace system.
       3) Define a number of remote queue definitions, (such as remote queue manager aliases and queue aliases) to allow WebSphere MQ messages to be routed onto the various WebSphere MQ Everyplace-bound transmission queues that you defined in step 10b1.
11. Modify your configuration on WebSphere MQ Everyplace to allow messages to pass from WebSphere MQ Everyplace to WebSphere MQ:
    a. Publish details about all the queue managers on your WebSphere MQ network you want to send messages to from the WebSphere MQ Everyplace network. Each WebSphere MQ queue manager requires a connections definition on your WebSphere MQ Everyplace server. All fields except the Queue manager name should be null, to indicate that the normal WebSphere MQ Everyplace communications connections should not be used to talk to this queue manager.
    b. Publish details about all the queues on your WebSphere MQ network you want to send messages to from the WebSphere MQ Everyplace network. Each WebSphere MQ queue requires a WebSphere MQ bridge queue definition on your WebSphere MQ Everyplace server. (This is the WebSphere MQ Everyplace equivalent of a `DEFINE QREMOTE` in WebSphere MQ).
       - The queue name is the name of the WebSphere MQ queue to which the bridge should send any messages arriving on this WebSphere MQ bridge queue.
       - The queue manager name is the name of the WebSphere MQ queue manager on which the queue is located.
       - The bridge name indicates which bridge should be used to send messages to the WebSphere MQ network.

- The WebSphere MQ queue manager proxy name is the name of the WebSphere MQ queue manager proxy object, in the WebSphere MQ Everyplace configuration, that can connect to a WebSphere MQ queue manager.
- The WebSphere MQ queue manager should have a route defined to allow messages to be posted to the `Queue Name` on `Queue Manager Name` to deliver the message to its final destination.

12. Start your WebSphere MQ and WebSphere MQ Everyplace systems to allow messages to flow. The WebSphere MQ system client channel listener must be started. All the objects you have defined on the WebSphere MQ Everyplace must be started. The simplest way to start objects manually, is to send a **start** command to the relevant bridge object. This command should indicate that all the children of the bridge, and children's children should be started as well.
    - To allow messages to pass from WebSphere MQ Everyplace to WebSphere MQ, start the client connection objects in WebSphere MQ Everyplace.
    - To allow messages to pass from WebSphere MQ to WebSphere MQ Everyplace, start both the client connection objects, and the relevant transmission queue listeners.
13. Create transformer classes, and modify your WebSphere MQ Everyplace configuration to use them. A transformer class converts messages from WebSphere MQ message formats into an WebSphere MQ Everyplace message format, and vice-versa. These format-converters must be written in Java and configured in several places in the WebSphere MQ bridge configuration.
    a. Create transformer classes
        - Determine the message formats of the WebSphere MQ messages that need to pass over the bridge.
        - Write a transformer class, or a set of transformer classes to convert each WebSphere MQ message format into an WebSphere MQ Everyplace message. Transformers are not directly supported by the C Bindings. See*WebSphere MQ Everyplace Application Programming Guide* for information about writing transformers in Java.
    b. You can replace the default transformer class. Use the administration GUI to **update** the default transformer class parameter in the bridge object's configuration.
    c. You can specify a non-default transformer for each WebSphere MQ bridge queue definition. Use the administration GUI to **update** the *`transformer`* field of each WebSphere MQ bridge queue in the configuration.
    d. You can specify a non-default transformer for each WebSphere MQ transmission queue listener. Use the administration GUI to **update** the *`transformer`* field of each listener in the configuration.
    e. Restart the bridge, and listeners.

Configuration example

This section describes an example configuration of 4 systems.

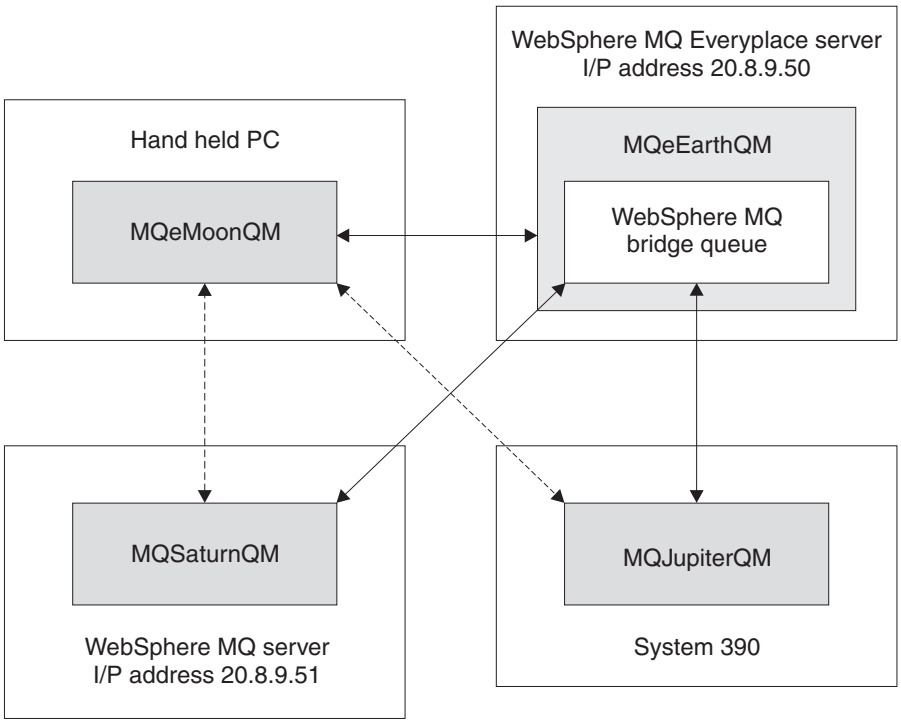

Figure 20. Configuration example

The four systems are:

MQeMoonQM
This is an WebSphere MQ Everyplace client queue manager, sited on a handheld PC. The user periodically attaches the handheld PC to the network, to communicate with the MQeEarthQM WebSphere MQ Everyplace gateway.

MQeEarthQM
This is on a Windows/2000 machine, with an I/P address of 20.8.9.50 This is an WebSphere MQ Everyplace gateway (server) queue manager.

MQSaturnQM
This is a WebSphere MQ queue manager, installed on a Windows/NT platform. The I/P address is 20.8.9.51

MQJupiterQM
This is a WebSphere MQ 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:

1. Define a **connection** with the following parameters:

   Target queue manager name: MQeEarthQM
   Adapter: FastNetwork:20.8.9.50

   Note: Check that the adapter you specify when you define the connection matches the adapter used by the Listener on the MQeEarthQM queue manager.

   Applications can now connect directly to any queue defined on the MQeEarthQM queue manager directly, when the MQeMoonQM is connected to the network. The requirement states that applications on MQeMoonQM must be able to send messages to MQeEarthQ in an asynchronous manner. This requires a remote queue definition to set up the asynchronous linkage to the MQeEarthQ queue.
2. Define a **remote queue** with the following parameters:

   Queue name: MQeEarthQ
   Queue manager name: MQeEarthQM
   Access mode: Asynchronous

   Applications 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, use a store-and-forward queue on the MQeEarthQM to collect messages destined for the MQeMoonQM queue manager.

On MQeEarthQM:

1. Define a **store-and-forward-queue** with the following parameters:

   Queue name: TO.HANDHELDS
   Queue Manager Name: MQeEarthQM
2. Add a **queue manager** to the **store-and-forward queue** using the following parameters:

   Queue Name: TO.HANDHELDS
   Queue manager: MQeMoonQM

A (similarly named) home-server queue is needed on the MQeMoonQM queue manager. This queue pulls messages out of the store-and-forward queue and puts them to a queue on the MQeMoonQM queue manager.

On MQeMoonQM:

1. Define a **home-server queue** with the following parameters:

Queue Name: TO.HANDHELDS
Queue manager name: MQeEarthQM

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 messages to MQeMoonQ in an asynchronous manner.

Enabling MQeEarthQM to send a message to MQSaturnQ

On MQeEarthQM:

1. Define a **bridge** with the following parameters:

   Bridge name: MQeEarthQMBridge
2. Define an **WebSphere MQ queue manager proxy** with the following parameters:

   Bridge Name: MQeEarthQMBridge
   MQ QMgr Proxy Name: MQSaturnQM
   Hostname: 20.8.9.51
3. Define a **client connection** with the following parameters:

   Bridge Name: MQeEarthQMBridge
   MQ QMgr Proxy Name: MQSaturnQM
   ClientConnectionName: MQeEarth.CHANNEL
   SyncQName: MQeEarth.SYNC.QUEUE
4. Define a **connection** with the following parameters:

   ConnectionName: MQSaturnQM
   Channel: null
   Adapter: null
5. Define an **WebSphere MQ bridge queue** with the following parameters:

   Queue Name: MQSaturnQ
   MQ Queue manager name: MQSaturnQM
   Bridge name: MQeEarthQMBridge
   MQ QMgr Proxy Name: MQSaturnQM
   ClientConnectionName: MQeEarth.CHANNEL

On MQSaturnQM:

1. Define a **server connection channel** with the following parameters:

   Name: MQeEarth.CHANNEL
2. Define a **local sync queue** with the following parameters:

   Name: MQeEarth.SYNC.QUEUE

   The sync queue is needed for assured delivery.

Applications on MQeEarthQM can now send messages to the MQSaturnQ on MQSaturnQM.

Enabling MQeEarthQM to send a message to MQJupiterQ

On MQeEarthQM:

1. Define a **connection** with the following parameters:

   ConnectionName: MQeJupiterQM
   Channel: null
   Adapter: null
2. Define an **WebSphere MQ bridge queue** with the following parameters:

   Queue Name: MQJupiterQ
   MQ Queue manager name: MQJupiterQM
   Bridge name: MQeEarthQMBridge
   MQ QMgr Proxy Name: MQSaturnQM
   ClientConnectionName: MQeEarth.CHANNEL

On MQSaturnQM:

1. Define a **remote queue definition** with the following parameters:

   Queue Name: MQJupiterQ
   Transmission Queue: MQJupiterQM.XMITQ

On both MQSaturnQM and MQJupiterQM:

1. Define a **channel** to move the message from the MQJupiterQM.XMITQ on MQSaturnQM to MQJupiterQM.

Now applications on MQeEarthQM can send a message to MQJupiterQ on MQJupiterQM, through MQSaturnQM.

Enabling MQeMoonQM to send a message to MQJupiterQ and MQSaturnQ

On MQeMoonQM:

1. Define a **connection** with the following parameters:

   Target Queue manager name: MQSaturnQM
   Adapter: MQeEarthQM

   The connection indicates that any message bound for the MQSaturnQM queue manager should go through the MQeEarthQM queue manager.
2. Define a **remote queue definition** with the following parameters:

   Queue name: MQSaturnQ
   Queue manager name: MQSaturnQM
   Access mode: Asynchronous
3. Define a **connection** with the following parameters:

   Target Queue manager name: MQJupiterQM
   Adapter: MQeEarthQM
4. Define a **remote queue definition** with the following parameters:

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:

1. Define a **local queue** with the following parameters:

   Queue name: MQeEarth.XMITQ
   Queue type: transmission queue
2. Define a **queue manager alias** (remote queue definition) with the following parameters:

   Queue name: MQeEarthQM
   Remote queue manager name: MQeEarthQM
   Transmission queue: MQeEarth.XMITQ

On MQeEarthQM:

1. Define a **Transmission queue listener** with the following parameters:

   Bridge name: MQeEarthQMBridge
   MQ QMgr Proxy Name: MQSaturnQM
   ClientConnectionName: MQeEarth.CHANNEL
   Listener Name: MQeEarth.XMITQ

Applications on MQSaturnQM can now send messages to MQeEarthQ using the MQeEarthQM queue manager alias . This routes each message onto the MQeEarth.XMITQ, where the WebSphere MQ Everyplace transmission queue listener MQeEarth.XMITQ gets them, and moves them onto the WebSphere MQ Everyplace network.

Enabling MQSaturnQM to send messages to the MQeMoonQ

On MQSaturnQM:

1. Define a **queue manager alias** (remote queue definition) with the following parameters:

   Queue name: MQeMoonQM
   Remote queue manager name: MQeMoonQM
   Transmission queue: MQeEarth.XMITQ

Applications on MQSaturnQM can now send messages to MQeMoonQ using the MQeMoonQM queue manager alias . This routes each message to the MQeEarth.XMITQ, where the WebSphere MQ Everyplace transmission queue listener MQeEarth.XMITQ gets them, and posts them onto the WebSphere MQ 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 WebSphere MQ routing techniques.

Now any application connected to any of the queue managers can post a message to any of the queues `MQeMoonQ`, `MQeEarthQ`, `MQSaturnQ` or `MQJupiterQ`.

Administration of the WebSphere MQ bridge

This section contains information on the tasks associated with the administration of the WebSphere MQ bridge

WebSphere MQ 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 queried or updated.

If the WebSphere MQ bridge queue references a bridge administered object that is 'stopped', it is unable to convey an WebSphere MQ Everyplace message onto the WebSphere MQ network until the bridge, WebSphere MQ queue manager proxy, and client connection objects are all 'started'.

The run state of administered objects can be changed using the **start** and **stop** actions from the MQeMQBridgeAdminMsg, MQeMQQMgrProxyAdminMsg, MQeClientConnectionAdminMsg or MQeListenerAdminMsg administration message classes.

The actions supported by the WebSphere MQ 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 its 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 its run-state. For example, sending **start** to a bridge object with *`affect 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 object cannot be started unless its parent object has already been started. Sending a start event to an administered object attempts to start all the objects higher in the hierarchy that 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 an administered object.

If the administered object is running, 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 changes 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".)

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`, 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`.

WebSphere MQ bridge considerations when shutting down a WebSphere MQ queue manager

We recommend that before you stop a WebSphere MQ queue manager, you issue a **stop** administration message to all the WebSphere MQ queue-manager-proxy bridge objects. This stops the WebSphere MQ Everyplace network from trying to use the WebSphere MQ queue manager and possibly interfering with the shutdown of the WebSphere MQ queue manager. (This can also be achieved by issuing a single **stop** administration message to the MQebridges object.)

If you choose not to stop the WebSphere MQ queue-manager-proxy bridge object before you shut the WebSphere MQ queue manager, the behavior of the WebSphere MQ shutdown and the WebSphere MQ bridge depends on the type of WebSphere MQ queue manager shutdown you choose, immediate shutdown or controlled shutdown.

Immediate shutdown
Stopping a WebSphere MQ queue manager using immediate shutdown severs any connections that the WebSphere MQ bridge has to the WebSphere MQ queue manager (this applies to connections formed using the MQSeries Classes for Java in either the bindings or client mode). The WebSphere MQ system shuts down as normal.

This causes all the WebSphere MQ bridge transmission queue listeners to stop immediately, each one warning that it has shut down due to the WebSphere MQ queue manager stop.

Any WebSphere MQ bridge queues that are active retain a (broken) connection to the WebSphere MQ queue manager until:

- The connection times-out, after being idle for an idle time-out period (as specified on the client-connection bridge object), at which point the broken connection is closed.
- The WebSphere MQ bridge queue is told to perform some action, such as put a message to WebSphere MQ, that attempts to use the broken connection. The **putMessage** operation fails and the broken connection is closed.

When an WebSphere MQ bridge queue has no connection, the next operation on that queue causes a new connection to be obtained. If the WebSphere MQ queue manager is not available, the operation on the queue fails synchronously. If the WebSphere MQ queue manager has been restarted after the shutdown, and a queue operation, such as **putMessage**, acts on the bridge queue, then a new connection to the active WebSphere MQ queue manager is established, and the operation executes as expected.

Controlled shutdown

Stopping a WebSphere MQ queue manager using the controlled shutdown does not sever any connections immediately, but waits until all connections are closed (this applies to connections formed using the MQSeries Classes for Java in either the bindings or client mode). Any active WebSphere MQ bridge transmission queue listeners notice that the WebSphere MQ system is quiescing, and stop with a relevant warning.

Any WebSphere MQ bridge queues that are active retain a connection to the WebSphere MQ queue manager until:

- The connection times-out, after being idle for an idle time-out period (as specified on the client connection bridge object), at which point the broken connection is closed, and the controlled shutdown of the WebSphere MQ queue manager completes.
- The WebSphere MQ bridge queue is told to perform some action, such as put a message to WebSphere MQ, that attempts to use the (broken) connection. The `putMessage` operation fails, the broken connection is closed, and the controlled shutdown of the WebSphere MQ 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 WebSphere MQ client connections until the connection idle time exceeds the idle time-out period (as specified on the client connection object configuration), at which point the connections in the pool are closed.

When the last client connection to the WebSphere MQ queue manager is closed, the WebSphere MQ controlled shutdown completes.

Administered objects and their characteristics

This section describes the characteristics of the different types of administered objects associated with the WebSphere MQ Everyplace WebSphere MQ 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 message 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 93. The label constants are defined in the header file published/MQe_MQBridge_Constants.h. If you include published/MQe_API.h in you installation, this file is included automatically.

Characteristics of bridges objects

- Run-state
- Children
- Child

Characteristics of bridge objects

- Description
- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- HeartBeatInterval
- DefaultTransformer

Characteristics of WebSphere MQ queue manager proxy objects

- Description
- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- HostName

Characteristics of client connection objects

- Description
- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- ClientConnectionName
- Port
- AdapterClass
- MQUserID
- MQPassword
- SendExit
- ReceiveExit

- SecurityExit
- CCSID
- SyncQName
- SyncQPurgerRulesClass
- MaxConnectionIdleTime
- SyncQPurgeInterval

Characteristics of WebSphere MQ transmission queue listener objects

- Description
- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- ClientConnectionName
- ListenerName
- DeadLetterQName
- ListenerStateStoreAdapter
- UndeliveredMessageRuleClass
- TransformerClass

Attribute details

AdapterClass

Type: Unicode

Label:

`MQE_FIELD_LABEL_ADAPTER`

Valid actions

Inquire, **create**, **update**

Description

Either a Java class name, or an alias that can be resolved into a Java class name. The client connection uses this attribute to determine which class to use to manipulate the WebSphere MQ system. Different versions of WebSphere MQ may recognize different adapter classes.

If this attribute is not specified, a default value of `com.ibm.mqe.mqbridge.MQeMQAdapter` is used.

This parameter is not validated

AdministeredObjectClass

Type: Unicode

Label:

`MQE_FIELD_LABEL_ADMINISTERED_OBJECT`

Valid actions

Inquire, **create**, **update**

Description

The name of a Java class, or an alias that resolves into a Java class name using the WebSphere MQ Everyplaceclass aliasing technique. The '.class' extension is not required.

This attribute describes the Java class that is used to provide the function for the bridge administered object. The value set depends on the type of administered object being configured.

This field should not be set, or changed without detailed instructions from IBM staff.

If this parameter is not specified when the object is created, it defaults to the following allowable values:

Object being configured	Value
A bridge object	com.ibm.mqe.mqbridge.MQeMQBridge
WebSphere MQ queue manager proxy object	com.ibm.mqe.mqbridge.MQeMQQMgrProxy
WebSphere MQ client connection object	com.ibm.mqe.mqbridge.MQeClientConnection
WebSphere MQ transmission queue listener object	com.ibm.mqe.mqbridge.MQeListener

Valid characters are: "0–9" "A-Z" "a-z" - . % /

BridgeName

Type: Unicode

Label:

`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 can be set only once, with the **create** administration message. Thereafter it is used to identify which bridge administered object should be inquired on, updated, deleted, started, or stopped.

CCSID

Type: Int

Label:

`MQE_FIELD_LABEL_CCSID`

Valid actions

Inquire, **create**, **update**

Description
See the WebSphere MQ Using Java documentation for a description of this parameter.

Valid values are: 0 to MAXINT. The default is 0.

Child

Type: Unicode

Label:

MQE_FIELD_LABEL_CHILD

Valid actions
Inquire

Description
A field containing the name of an WebSphere MQ bridge administered object.

Children

Type: MQeFields array

Label:

MQE_FIELD_LABEL_CHILDREN

Valid actions
Inquire

Description
An array of Child fields, each element containing a Child attribute.

ClientConnectionName

Type: Unicode

Label:

MQE_FIELD_LABEL_CLIENT_CONNECTION_NAME

Valid actions
Inquire, **create**, **update**, **delete**, **start**, **stop**

Description

Note: This characteristic can be set only once, with the **create** administration message. Thereafter it is used to identify which bridge administered object should be inquired on, updated, deleted, started, or stopped.

DeadLetterQName

Type: Unicode

Label:

MQE_FIELD_LABEL_DEAD_LETTER_Q_NAME

Valid actions
Inquire, **create**, **update**

Description
If the gateway finds it cannot deliver a message from WebSphere MQ to WebSphere MQ Everyplace, the message cannot be processed by the gateway, and it is placed on a dead letter queue on the WebSphere MQ system. This parameter defines which queue the erroneous message is delivered to.

The default value is SYSTEM.DEAD.LETTER.QUEUE.

DefaultTransformer

Type: Unicode

Label:

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 WebSphere MQ to WebSphere MQ Everyplace, the target queue may have a transformer class defined. If a transformer is not defined, this class is used to transform the WebSphere MQ message into the WebSphere MQ Everyplace format.

When a message is sent from WebSphere MQ Everyplace to WebSphere MQ, the transmission queue listener moving the message onto WebSphere MQ Everyplace may have a transformer class defined. If a transformer is not defined, this class is used to transform the WebSphere MQ Everyplace message into the WebSphere MQ format.

No validation of the value in this field is performed.

The default value is com.ibm.mqebridge.MQeBaseTransformer

Description

Type: Unicode

Label:

MQE_FIELD_LABEL_DESCRIPTION

Valid actions

Inquire, **create**, **update**

Description

A free-format Unicode string, used by an administrator to describe the configured object. WebSphere MQ Everyplace does not use the contents of this field. The contents of this field are not validated by WebSphere MQ Everyplace.

HeartBeatInterval

Type: Int

Label:

MQE_FIELD_LABEL_HEARTBEAT_INTERVAL

Valid actions

Inquire, **create**, **update**

Description

A time interval, expressed in units of 1 minute, with values between 1 and 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 WebSphere MQ connections' or 'the sync queue will be purged'. The heartbeat

provides an interval for the timers that is indivisible. The lower this value is set, the more accurate any timer related actions will be. For instance, if you say 'reap all WebSphere MQ connections if they have been idle for more than 10 minutes', but set the heartbeat interval tor 3 minutes, then an idle WebSphere MQ connection is checked after 3,6,9 and 12 minutes, but is only 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.

Hostname

Type: Unicode

Label:

`MQE_FIELD_LABEL_HOST_NAME`

Valid actions

Inquire, **create**, **update**

Description

Used to create connections to this WebSphere MQ queue manager using the WebSphere MQ Classes for Java. If this characteristic is not specified, the WebSphere MQ queue manager is assumed to be on the same machine as the JVM, so the Java bindings mode is used to communicate with the WebSphere MQ system.

Note: A blank value is *not* the same as specifying `localhost`. If a blank value is used, then the WebSphere MQ bridge uses the WebSphere MQ Classes for Java in bindings mode which communicates directly with WebSphere MQ. If you specify `localhost`, the WebSphere MQ bridge uses the WebSphere MQ Classes for Java in client mode. This means that all communication with WebSphere MQ is through the network (TCP/IP) stack.

The value specified here is not validated. 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 use the I/P address notation in this field if a symbolic name causes problems.

ListenerName

Type: Unicode

Label:

`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 WebSphere MQ 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 can be set only once, with the **create** administration message. Thereafter it is used to identify which WebSphere MQ bridge administered object should be inquired on, updated, deleted, started, or stopped.

ListenerStateStoreAdapter

Type: Unicode

Label:

`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 file system.
- com.ibm.mqe.mqbridge.MQeMQAdapter, which stores state information on the WebSphere MQ server.

The disk adapter is generally quicker than using the WebSphere MQ-based adapter. The classname can be followed by a colon separated list of arguments, although only the MQeDiskFieldsAdapter uses them. In this case the 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 currently exist. If the supplied path name ends in a folder separator, for example "\" in DOS. 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`.

MaxConnectionIdleTime

Type: Int

Label:

`MQE_FIELD_LABEL_MAX_CONNECTION_IDLE_TIME`

Valid actions

Inquire, **create**, **update**

Description

Each client connection object in the bridge maintains a pool of WebSphere MQ Java client connections to its WebSphere MQ system.

When a WebSphere MQ connection becomes idle through lack of use, 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 saves resources when the connection is idle. The connection pool is an efficiency device that is used within the WebSphere MQ bridge. The creation of new WebSphere MQ client connections is a resource intensive operation. If there are idle connections in the pool, one of these is reused, thus avoiding a creation operation. The higher the *MaxConnectionIdleTime* value, the more likely it is that an idle connection will be waiting in the pool, but idle client connections consume 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: Between 0 and 720 (12 hours). The default is 5 (minutes).

Setting this value to 0 is not recommended as it effectively means 'don't use a connection pool', and whenever a WebSphere MQ client connection is idle, it is reaped or discarded.

This time-out is only checked at the interval set by the *heartbeatInterval* parameter.

MaxConnectionIdleTime can have a direct effect on the length of time it takes to shut down an WebSphere MQ Everyplace system. Refer to
"WebSphere MQ bridge considerations when shutting down a WebSphere MQ queue manager" on page 90 for more details.

MQPassword

Type: Unicode

Label:

`MQE_FIELD_LABEL_PASSWORD`

Valid actions

Inquire, **create**, **update**

Description

Used by the MQSeries Classes for Java. If this attribute is not specified, the password field on the WebSphere MQ calls is set to "" . The value you specify here overrides any defaults. This parameter is not validated.

MQQMgrProxyName

Type: Unicode

Label:

`MQE_FIELD_LABEL_MQ_Q_MGR_PROXY_NAME`

Valid actions

Inquire, **create**, **update**, **delete**, **start**, **stop**

Description
The name of the queue manager proxy object. (In other words, the name of the target WebSphere MQ queue manager.)

Note: This characteristic can be set only once, with the **create** administration message. Thereafter it is used to identify which bridge administered object should be inquired on, updated, deleted, started, or stopped.

MQUserID

Type: Unicode

Label:

`MQE_FIELD_LABEL_USER_ID`

Valid actions
Inquire, **create**, **update**

Description
Used by the MQSeries Classes for Java. If this parameter is not specified. the user-id field on the WebSphere MQ calls is set to "" . The value you specify here overrides any defaults. This parameter is not validated.

Port

Type: Int

Label:

`MQE_FIELD_LABEL_PORT`

Valid actions
Inquire, **create**, **update**

Description
Used to create connections to this WebSphere MQ queue manager using the WebSphere MQ Classes for Java. If this parameter is not specified, the WebSphere MQ queue manager is assumed to be on the same machine as the JVM. In this case, the bindings mode of the WebSphere MQ Classes for Java is used to communicate with the WebSphere MQ system.

Valid range 0 to `MAXINT`.

ReceiveExit

Type: Unicode

Label:

`MQE_FIELD_LABEL_RECEIVE_EXIT`

Valid actions
Inquire, **create**, **update**

Description
Used to match the exit used at the other end of the Client connection.

This parameter is not validated.

Run-state

Type: Int

Label:

MQE_FIELD_LABEL_RUN_STATE

Valid actions

Inquire

Description

Indicates whether the administered object is running (value=1), or stopped (value=0). When an object is stopped it can have its properties changed.

SecurityExit

Type: Unicode

Label:

MQE_FIELD_LABEL_SECURITY_EXIT

Valid actions

Inquire, create, update

Description

Used to match the exit used at the other end of the Client connection.

This parameter is not validated.

SendExit

Type: Unicode

Label:

MQE_FIELD_LABEL_SEND_EXIT

Valid actions

Inquire, **create**, **update**

Description

Used to match the exit used at the other end of the Client connection.

This parameter is not validated.

StartupRuleClass

Type: Unicode

Label:

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 startup, or when the object is created. The rule class name is not validated.

The rule class 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 and all its parents to start. If this field is set to "" (blank) , the administered object is not started.Rules are not

directly supported by the C Bindings. See*WebSphere MQ Everyplace Application Programming Guide* for information about writing rules in Java.

SyncQName

Type: Unicode

Label:

`MQE_FIELD_LABEL_SYNC_Q_NAME`

Valid actions

Inquire, **create**, **update**

Description

The name of the sync queue on the WebSphere MQ queue manager that is used by the WebSphere MQ bridge . Valid characters forming the name are: "0-9" "A-Z" "a-z" _ . % / .The sync queue is a WebSphere MQ queue that is used to keep track of which messages are in the process of moving from WebSphere MQ Everyplace to WebSphere MQ. If a message is part way through the logic that assures the once-only delivery of a message, there is another message on the sync queue, indicating how far through the logic the message has progressed. If the WebSphere MQ Everyplace system is shut down cleanly, the sync queue should be empty. If the connection between the systems is broken, some persistent state information is left in the sync queue. The WebSphere MQ Everyplace system uses this information when it restarts and continues from where the process failed. 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 WebSphere MQ queue manager for WebSphere MQ Everyplace to WebSphere MQ message transfer to work. If the listener state class is the MQeMQAdapter, this sync queue is also used for storing persistent state information about the listeners. 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`.

SyncQPurgeInterval

Type: int

Label:

`MQE_FIELD_LABEL_SYNC_Q_PURGE_INTERVAL`

Valid actions

Inquire, **create**, **update**

Description

The time interval between successive purges of the sync queue, expressed in minutes.

When this interval elapses, the sync queue is scanned. If a message that has not been confirmed is found on the Sync queue, then the SyncQPurgerRules class is invoked to deal with this situation.

Zero indicates that the sync queue should never be purged.

The default is 60 minutes.

The actual granularity of the purging operation is dictated by the heartbeat-interval of the owning bridge definition. For example: If the hearbeat interval is set to 10 minutes, but the purge interval is set to 9 minutes, then the purge operation occurs after 10 minutes. If, however, the purge interval is changed to 11 minutes, the purge operation still occurs after 10 minutes.

SyncQPurgerRulesClass

Type: Unicode

Label:

`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 WebSphere MQ Everyplace to confirm a message.

The default is a classname that just reports the condition in the WebSphere MQ Everyplace trace.

This parameter is not validated.

TransformerClass

Type: Unicode

Label:

`MQE_FIELD_LABEL_TRANSFORMER`

Valid actions

Inquire, **create**, **update**

Description

This is the name of the Java class that is used to convert the WebSphere MQ message into an WebSphere MQ Everyplace message. When a message is taken from WebSphere MQ by the listener, it is transformed into an WebSphere MQ Everyplace format message using the specified transformer. If the transformer class is specified as `null` or a blank string, then the *`DefaultTransformer`* parameter provided on the bridge configuration parameters is used as the transformer. If the default is also set to `null` or blank, messages cannot be transferred.

The default value is `""`

Transformers are not directly supported by the C Bindings. See*WebSphere MQ Everyplace Application Programming Guide* for information about writing transformers in Java.

UndelivedMessageRuleClass

Type: Unicode

Label:

`MQE_FIELD_LABEL_UNDELIVERED_MESSAGE_RULE_CLASS`

Valid actions

Inquire, **create**, **update**

Description

The name of the MQeUndeliveredMessageRule class. When a message moving from WebSphere MQ to WebSphere MQ 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`. Rules are not directly supported by the C Bindings. See*WebSphere MQ Everyplace Application Programming Guide* for information about writing rules in Java.

How to send a message from WebSphere MQ to WebSphere MQ Everyplace

There are many ways of arranging your routing on the WebSphere MQ system to test the transmission of a message. One method is to define queue manager aliases for each WebSphere MQ Everyplace queue manager that the bridge knows about. This document describes how to use the resultant configuration to send a message to the WebSphere MQ Everyplace queue.

1. Select the WebSphere MQ First Steps program from the WebSphere MQ Client v 5.1
2. Select the **API exerciser** from the 'First Steps' screen
3. In the 'API Exerciser Queue Managers' screen:
   - Select the WebSphere MQ queue manager to which the bridge is connected. (The example is called `MQA`)
   - Check the **Advanced mode** checkbox
   - Click the **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 that **MQOO_INPUT_AS_Q_DEF** is *not* selected
   - Make sure that **MQO_OUTPUT** *is* selected
   - Fill in the **ObjectName** field with the name of the queue that you wish the message to go to on the WebSphere MQ Everyplace queue manager. (The example is called `Q1`)
   - Fill in the **ObjectQMgrName field** with the name of the WebSphere MQ Everyplace queue manager that you wish the message to go to. (The example is called `ExampleQM`)
   - Click **OK** to open a route to the queue.
5. In the 'API Exerciser Queues' screen:
   - Click the **MQPUT** button to display the 'MQPUT -Argument Options' screen
6. In the 'MQPUT - Argument Options' screen:
   - Type in your message
   - Click **OK** to send the message to `Q1` on `ExampleQM` on the WebSphere MQ Everyplace system

Handling undeliverable messages

The WebSphere MQ bridge's transmission queue listener acts in a similar way to a WebSphere MQ connection, pulling messages from a WebSphere MQ transmission queue, and delivering them to the WebSphere MQ Everyplace network. It follows the WebSphere MQ 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 WebSphere MQ queue (on the *sending* queue manager).

Putting messages to the WebSphere MQ bridge queue

If an application uses **putMessage()**, specifying that a **confirmputMessage()** should not be used to confirm this message, the WebSphere MQ bridge does not use assured delivery logic to pass the message to WebSphere MQ. It does a simple **MQPut** to the target WebSphere MQ queue. If there is a failure anywhere along the message route, the application is unable to determine whether the message has been sent or not. If the application decides to resend the message, it is possible for two identical messages to arrive on the WebSphere MQ queue.

To avoid this problem, the application programmer should use a combination of **putMessage()** and **confirmputMessage()** calls. Using **putMessage()** with the *confirm* parameter set to `true` causes the WebSphere MQ bridge to use assured delivery logic to put the message to the WebSphere MQ system.

If any component of the path between the WebSphere MQ system and the sending application fails, the application is unable to determine whether the message got to its destination or not. In this case, the application should put the original message again, with a boolean MQeField added. For example:

```
C example
```

This indicates that this message has been sent in the past. The WebSphere MQ bridge uses its assured delivery logic to assure that only one of the two **putMessage()** calls actually put a message to WebSphere MQ.

If the **putMessage()** is used, with or without the *confirm* flag set, and a successful return code is received, the application can be sure that the message has been passed to the WebSphere MQ queue.

If the **putMessage()** is used, with the *confirm* flag set, the WebSphere MQ bridge retains some information about the message (on its sync queue) that enables it to prevent duplicate messages being sent by the application. The WebSphere MQ bridge can only prevent duplicate messages being sent if the *Qos_Retry* parameter is set. The **confirmputMessage()** removes the message history from the WebSphere MQ bridge sync queue.

The following procedure causes four messages to arrive on the target WebSphere MQ queue.

	create a new message	
(1)	`putMessage(Confirm=Yes)`	- Causes the message to be delivered to WebSphere MQ, but some note made on the sync queue.
	set the retry bit on the message	

	`putMessage(Confirm=Yes)`	- Suppressed, as the message is already noted in the sync queue.
	`putMessage(Confirm=Yes)`	- Suppressed, as the message is already noted in the sync queue.
(2)	`putMessage(Confirm=No)`	- *not* suppressed. The message is delivered to the WebSphere MQ queue.
	remove the retry bit from the message	
(3)	`putMessage(Confirm=Yes)`	- Causes the message to be sent to WebSphere MQ. The retry bit was not set, so the WebSphere MQ bridge did not look at its sync queue.
	`ConfirmputMessage()`	- Causes the WebSphere MQ bridge to clear its memory of the message.
	set the retry bit on the message	
(4)	`putMessage()`	- Causes the message to be sent.

Getting and browsing messages from the WebSphere MQ bridge queue

As with other WebSphere MQ Everyplace queues, it is possible to get and browse messages from WebSphere MQ bridge queues. It is also possible to specify an MQeFields filter on these operations. If a filter is used, the first message matching the filter is returned by **getMessage**) and all messages matching the filter are returned by **browseMessages**.

When browsing messages, if the filter field is blank or null, all messages are collected from the WebSphere MQ queue and are placed in the returning enumeration. If the filter is non-blank and non-null, all messages collected from the WebSphere MQ queue are passed through the queue's message transformer before being matched against the filter. Matching messages are placed in the returning enumeration.

If the filter field contains one or both of *MQE_MSG_MSGID* and *MQE_MSG_CORRELID*, messages are collected from the WebSphere MQ queue using one or both of the WebSphere MQ *MQE_MSG_MSGID* and *MQE_MSG_CORRELID* as filter elements. The results are then transformed into WebSphere MQ Everyplace messages which are filtered as follows:

1. The original filter is applied as the default match criteria and any matching messages are placed in the returning enumeration.
2. If any transformed WebSphere MQ Everyplace messages do not contain the *MQE_MSG_MSGID* field, the *MQE_MSG_MSGID* field is removed from the filter.
3. If any transformed WebSphere MQ Everyplace message do not contain the *MQE_MSG_CORRELID* field, the *MQE_MSG_CORRELID* field is removed from the filter.
4. The unmatched *`MQSeries Everyplace`* messages are then filtered using the new filter, and matching messages are placed into the returning enumeration.

Note that using a blank or null filter, or a filter that contains neither the *MQE_MSG_MSGID* field nor the *MQE_MSG_CORRELID* field causes all messages on the WebSphere MQ queue to be browsed. To optimize performance, try to include in the filter one or both of the expected *`Message Id`* or *`CorrelId`* as it exists in WebSphere MQ format.

Filters on **getMessage** work in a similar way to filters on **browseMessages**, except that only the first match is removed from the WebSphere MQ queue and returned to the application.

Usage restrictions

There are some restrictions on the use of **getMessage** and **browseMessages** with WebSphere MQ bridge queues:

- It is not possible to get or browse messages from WebSphere MQ bridge queues that point to WebSphere MQ remote queue definitions.
- You cannot use a nonzero *Confirm ID* on WebSphere MQ bridge queue gets. This means that the **getMessage** operation on WebSphere MQ bridge queues does not provide assured delivery. If you need a get operation to be assured, you should use transmission-queue listeners to transfer messages from WebSphere MQ.
- Because messages originating from WebSphere MQ do not contain unique identifiers, it is not possible to browse messages using the *justUID* flag set to true (this would normally return a list of the unique message identifiers that matched the browse).
- The **browseMessagesAndLock()** function is not supported.

National language support implications

This section describes how the WebSphere MQ bridge handles messages flowing between MQSeries systems that use different national languages. The diagram in Figure 21is used to describe the flow of a message from an WebSphere MQ Everyplace client application to a WebSphere MQ application.

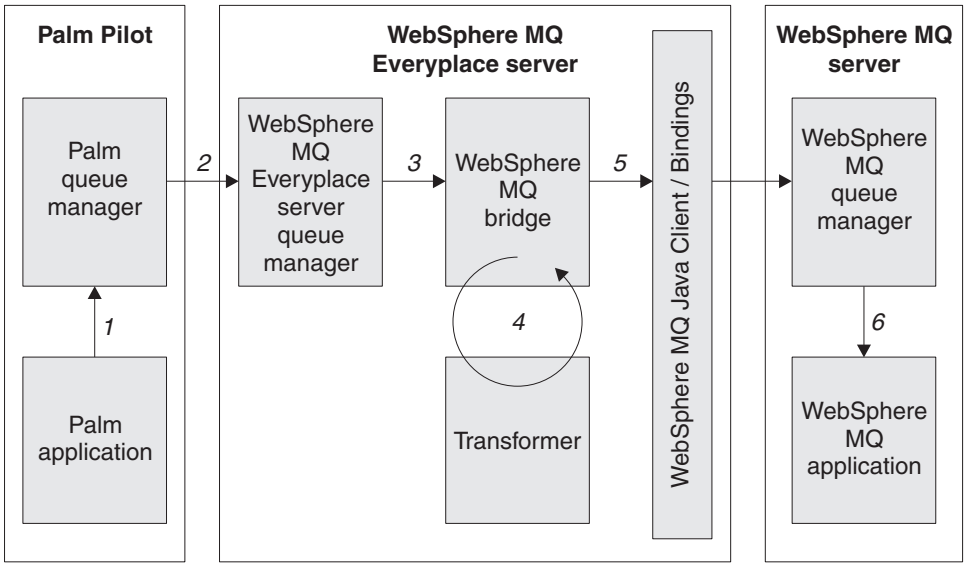

Figure 21. Message flow from WebSphere MQ Everyplace to WebSphere MQ

1. **Client application**
   a. The client application builds an WebSphere MQ Everyplace message object containing the following data:

      A Unicode field
      This string is generated using appropriate libraries available on the client machine (if C/C++ is being used).

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. **WebSphere MQ Everyplace server puts the message onto the WebSphere MQ bridge queue**

   The message is not translated at all through this step.

4. **WebSphere MQ bridge passes the WebSphere MQ Everyplace message to the user-written transformer**

   Note: The examples in this section are in Java because transformers can only be written in Java. (See "Unsupported Java APIs" on page 18

   The transformer creates a WebSphere MQ message as follows:

   - The Unicode field in the WebSphere MQ Everyplace message is retrieved using:

     ```
     String value = MQemsg.GetUnicode(fieldname)
     ```

   - The retrieved value is copied to the WebSphere MQ message using **MQmsg.writeChars(value)**
   - The byte field in the WebSphere MQ Everyplace message is retrieved using:

     ```
     Byte value = MQemsg.getByte(fieldName)
     ```

   - The retrieved value is copied to the WebSphere MQ message using **MQmsg.writeByte(value)**
   - The ascii field in the WebSphere MQ Everyplace message is retrieved using either **MQmsg.writeChars(value)** to create a unicode value, or **MQmsg.writeString(value)** to create a code-set-dependent value, in the WebSphere MQ message.

   If using **writeString()**, the character set of the string may also be set. The transformer returns the resultant WebSphere MQ message to the calling WebSphere MQ bridge code.

5. **The WebSphere MQ bridge passes the message to WebSphere MQ using the WebSphere MQ Classes for Java**

   Unicode values in the WebSphere MQ message are translated from big-endian to little-endian, and vice-versa, as required. Byte values in the WebSphere MQ message are translated from big-endian to little-endian, and vice-versa, as required. The field that was created using **writeString()** is translated as the message is put to WebSphere MQ, using conversion routines inside the WebSphere MQ Classes for Java. 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 WebSphere MQ Classes for Java client connection, and the CCSID of the receiving WebSphere MQ server connection.

6. **The message is got by a WebSphere MQ application**

   If the message contains a unicode string, the application must deal with that string as a unicode string, or else convert it into some other format (UTF8 for

example). If the message contains a byte string, the application may use the bytes as-is. (raw data). If the message contains a string, it is read from the message, and may be converted to a different data format as required by the application. This conversion is dependent on the codeset value in the *characterSet* header field.

Conclusion

If you have an WebSphere MQ Everyplace application, and wish to convey character-related data from WebSphere MQ Everyplace to WebSphere MQ, 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 character for a codepoint changes as you change between the various ASCII codepages) then you can use either **putUnicode** (which is never subject to translation between codepages), or **putArrayOfByte** (in which case you have to handle the translation between the sender's codepage and the receiver's 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 is never subject to translation between codepages) or **putAscii** (which is never subject to translation between codepages, as all your data lies within the invariant range of the ASCII codepages).

Chapter 8. Security

This section contains information about the security function provided by WebSphere MQ Everyplace. The different levels of security are described together with typical usage scenarios and usage guidance.

Security features

WebSphere MQ Everyplace provides an integrated set of security features that enable the protection of data when held locally and when it is being transferred. There are three different categories of security:

Local security
Local security provides protection for any WebSphere MQ Everyplace data.

Queue-based security
Queue-based security automatically protects WebSphere MQ Everyplace message data between the initiating queue manager and queue, on the queue, and between the queue and the receiving queue manager. 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 WebSphere MQ Everyplace application.

Queue based security is handled internally by WebSphere MQ Everyplace and does not require any specific action by the initiator or recipient of the message. Local and Message-level security must be initiated by an application.

All three categories protect Message data by the application of an MQeAttribute. 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 WebSphere MQ Everyplace security. Each category of security is described in detail later in this chapter.

WebSphere MQ Everyplace also provides the following services to assist with security:

Private registry services
WebSphere MQ 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. It also provides additional services so that functions can use the entity's private key, (for digital signature, and RSA decryption) without the private credentials leaving the PrivateRegistry instance.

These services are used by queue-based security and message-level security using MQeTrustAttribute.

Public registry services
WebSphere MQ Everyplace public registry provides a publicly accessible repository for mini-certificates.

These services can be used by queue-based and message-level security.

These services are described in more detail later in the chapter.

Local security

Local security provides protection for WebSphere MQ Everyplace data or MQeFields objects, including message, MQeMsg, objects. The protected data is returned in a byte array. To apply local security to a data object you must:

1. Create an attribute with an appropriate authenticator, cryptor, and compressor
2. Set up an appropriate *key* (by providing a password or passphrase seed)
3. Explicitly attach the key to the attribute, the attribute to the data (MQeFields) object, and invoke the **dump()** function on the data object

The authenticator determines how access to the data is controlled. The cryptor determines the cryptographic strength protecting the data confidentiality. The compressor determines the storage required by the message.

WebSphere MQ Everyplace provides the MQeLocalSecure class to assist with the use of local security. However, it is the responsibility of the local security user to setup an appropriate attribute and provide the password or passphrase key. MQeLocalSecure provides the function to protect the data and to save and restore it from backing storage. If an application chooses to attach an attribute to a message without using MQeLocalSecure, it also needs to save the data after using **dump** and must retrieve the data before using **restore**.

Usage scenario

Consider a scenario where mobile agents working on many different customer sites want to ensure that the confidential data of one customer is not accidentally shared with another. Local security features, using different keys, and possibly different cryptographic strengths, provide a simple function for protecting different customer data held on a single machine .

A simple extension of this scenario could be that the protected local data is accessed using a key that is pulled from a secure queue on an WebSphere MQ Everyplace server node. The agents client has to authenticate itself to access the server queue and pull the local key data, but never knows the actual key.

One of the advantages of taking this approach is that an audit trail is easily accumulated for all access to customer specific data.

Secure feature choices

When using MQeLocalSecure, the following attribute choices are available:

Authenticator
Example MQeWTLSCertAuthenticator

Cryptor
One of the symmetric cryptors MQeDESCryptor, MQe3DESCryptor, MQeRC4Cryptor, MQeRC6Cryptor or MQeMARSCryptor

Compressor
MQeLZWCompressor, MQeRleCompressor, or MQeGZIPCompressor

Selection criteria

You should use an authenticator if you need to provide additional controls to prevent access to the local data by unauthorized users. In some ways using an authenticator is unnecessary since providing the key password or passphrase automatically limits access to those who know this secret.

The choice of cryptor is driven by the strength of protection required. The stronger the encryption, the more difficulty an attacker would face when trying to get illegal access to the data. Data protected with symmetric ciphers that use 128 bit keys is acknowledged as more difficult to attack than data protected using ciphers that use shorter keys. However, in addition to cryptographic strength, the selection of a cryptor may also be driven by many other factors. An example is that some financial solutions require the use of triple DES in order to get audit approval.

You should use a compressor if you need to optimize the size of the protected data. However, the effectiveness of the compressor depends on the content of the data. The MQeRleCompressor performs run length encoding . This means that the compressor routines compress or expand repeated bytes. Hence it is effective in compressing and decompressing data with many repeated bytes. MQeLZWCompressor uses the LZW scheme. The simplest form of the LZW algorithm uses a dictionary data structure in which various words (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). The MQeGZIPCompressor uses the same compression algorithm as the **gzip** command on UNIX. This searches for repeating patterns in the data and replaces subsequent occurrences of a pattern with a reference back to the first occurrence of the pattern.

Usage guide

1. The following pseudo-code protects an MQeFields object using MQeLocalSecure

```
/*SIMPLE PROTECT FRAGMENT */
   #define BUF_LENGTH  1000  /* assuming this is big enough */

   MQeDESCryptorHndl desC;
   MQeAttributeHndl attr;
   MQeKeyHndl  localkey;
   MQeFieldsHndl  myData;
   MQEBYTE   protectedData[BUF_LENGTH];
   MQEINT32  bufLength = BUF_LENGTH;
   MQERETURN  rc;
   MQeExceptBlock exceptBlock;

   /*instantiate a DES cryptor */
   rc = mqeDESCryptor_new(&exceptBlock, &desC);
               /*instantiate an Attribute using the DES cryptor */
   rc = mqeAttribute_new(&exceptBlock, &attr, NULL,
          NULL, desC, NULL);
   /*instantiate a base Key object */
   rc = mqeKey_new(&exceptBlock, &localKey);
   /*set the base Key object local key */
   rc = mqeKey_setLocalKey(localKey, &exceptBlock,
            MQeString("my secret key"));
   /*attach the key to the attribute */
   rc = mqeAttribute_setKey(attr, &exceptBlock, localkey);
   /*instantiate a MQeFields object */
                rc = mqeFields_new(&exceptBlock, &myData);
   /*attach the attribute to the data object */
```

```
mqeFields_setAttribute(myData, &exceptBlock, attr);
/*add some test data */
rc = mqeFields_putAscii(myData, &exceptBlock, MQeString("testdata"),
                            MQeString("0123456789abcdef...."));
/*encode the data */
rc = mqeFields_dump(myData, &exceptBlock, protectedData,
          &bufLength, MQE_FALSE);

/* SIMPLE UNPROTECT FRAGMENT                                    */
#define BUF_LENGTH  1000  /* assuming this is big enough */

MQeDESCryptorHndl desC2;
MQeAttributeHndl attr2;
MQeLocalSecureHndl ls2;
MQEBYTE   outBuf[BUF_LENGTH];
MQEINT32  bufLength = BUF_LENGTH;
MQERETURN  rc;
MQeExceptBlock exceptBlock;

/* instantiate a DES cryptor                                    */
rc = mqeDESCryptor_new(&exceptBlock, &desC2);
/* instantiate an attribute using the DES cryptor               */
              rc = mqeAttribute_new(&exceptBlock,
           &attr2, NULL, NULL, des2C, NULL);
/* instantiate a (a helper) LocalSecure object                  */
              rc = mqeLocalSecure_new(&exceptBlock, &ls2);
/* open LocalSecure obj identifying target file and directory */
rc = mqeString_newChar8(&exceptBlock, &fileDir, ".\\");
rc = mqeLocalSecure_open(ls2,  &exceptBlock, MQeString(".\\"),
                               MQeString("TestSecureData.txt"));

/* use LocalSecure read to restore from target and decode data */
rc = mqeLocalSecure_read(ls2, &exceptBlock, outBuf, &bufLength,
                              attr2, MQeString("my secret key"));
```

2. The following pseudo-code protects an MQeFields locally without using MQeLocalSecure.

```
/*SIMPLE PROTECT FRAGMENT */
   #define BUF_LENGTH  1000  /* assuming this is big enough */

   MQeDESCryptorHndl desC;
   MQeAttributeHndl attr;
   MQeKeyHndl  localkey;
   MQeFieldsHndl  myData;
   MQEBYTE   protectedData[BUF_LENGTH];
   MQEINT32  bufLength = BUF_LENGTH;
   MQERETURN  rc;
   MQeExceptBlock exceptBlock;

   /*create a DES cryptor */
   rc = mqeDESCryptor_new(&exceptBlock, &desC);
               /*create an Attribute using the DES cryptor */
   rc = mqeAttribute_new(&exceptBlock,&attr, NULL, NULL, desC, NULL);
   /*create a base Key object */
   rc = mqeKey_new(&exceptBlock, &localKey);
   /*set the base Key object local key */
   rc = mqeKey_setLocalKey(localKey, &exceptBlock,
            MQeString("my secret key"));
   /*attach the key to the attribute */
   rc = mqeAttribute_setKey(attr, &exceptBlock, localkey);
   /*create a MQeFields object */
               rc = mqeFields_new(&exceptBlock, &myData);
   /*attach the attribute to the data object */
   mqeFields_setAttribute(myData, &exceptBlock, attr);
   /*add some test data */
   rc = mqeFields_putAscii(myData, &exceptBlock,
                               MQeString("testdata"),
```

```
         MQeString("0123456789abcdef...."));
/*encode the data */
rc = mqeFields_dump(myData, &exceptBlock, protectedData,
       &bufLength, MQE_FALSE);

/* SIMPLE UNPROTECT FRAGMENT                                    */
#define BUF_LENGTH  1000  /* assuming this is big enough */

MQeDESCryptorHndl desC2;
MQeAttributeHndl attr2;
MQeKeyHndl  localKey2;
MQeFieldsHndl  myData2;
MQEBYTE   protectedData[BUF_LENGTH];
MQEINT32  bufLength = BUF_LENGTH;
MQERETURN  rc;
MQeExceptBlock exceptBlock;

/* read protected data into protectedData and set
 bufLength to the data length */
...
/* create a DES cryptor  */
rc = mqeDESCryptor_new(&exceptBlock, &desC2);
/*create an attribute using the DES cryptor             */
              rc = mqeAttribute_new(&exceptBlock, &attr2,
                                    NULL, NULL, des2C, NULL);
/* create a (a helper) LocalSecure object               */
              rc = mqeKey_new(&exceptBlock, &localKey);
/*set the base Key object local key */
rc = mqeKey_setLocalKey(localKey2, &exceptBlock,
         MQeString("my secret key"));
        /*attach the key to the attribute */
mqeAttribute_setKey(attr2, &exceptBlock, localkey2);
 /*create a new data object */
rc = mqeFields_new(&exceptBlock, &myData);
/*attach the attribute to the data object */
mqeFields_setAttribute(myData2, &exceptBlock, attr2);
/*decode the data */
mqeFields_restore(myData2, &exceptBlock, protectedData,
       bufLength, MQE_FALSE);
```

Queue-based security

Queue-based security automatically protects WebSphere MQ Everyplace message data between the initiating queue manager and the queue, on the queue itself, and between the queue and the receiving queue manager. This form of protection requires the target queue to be defined with an attribute. This protection is independent of whether the 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 authenticator, 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 authenticator before the operation is permitted. If the operation is permitted, the message data is automatically encoded and decoded using the attribute's MQe3DESCryptor and MQeRleCompressor. 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. This protection applies during transfer between the initiating queue manager and the queue, while the message is stored on the queue, and during transfer between the queue and the receiving queue manager.

Queue-based security and asynchronous queues

With synchronous queues, queue-based security is relatively simple. In this case a message is put to a synchronous remote queue definition that has the same security attributes as the destination queue. The message is transmitted over a connection with appropriate security attributes and is stored on the secure queue.

With asynchronous queues, especially Store-and-forward queues and Home-server queues, the transmitting and receiving queues are more likely to have different security attributes. These differences have to be managed during message transfer.

Once a message has been put to an asynchronous queue it is transmitted from one queue to another until it reaches its destination. A queue manager is responsible for requesting the transfer of the message between a pair of queues and another queue manager is responsible for responding to the request.

If queue based security is used, the requesting queue manager tries to establish a connection with security attributes that match the queue that it owns. The queue manager receiving the request checks that the existing attributes are sufficient for its queue. For example, suppose a client queue manager has a queue with a DES cryptor on it and messages are routed from this to a server's Store-and-forward queue that has a MARS cryptor. When the client is triggered to send a message it tries to establish a DES encrytped connection to the server; the server asks the Store-and-forward queue whether it will accept messages over a DES encrypted connection.

If a queue doesn't have any attribute rules, it only accepts a connection that has exactly the same cryptor as itself. This behavior can be overridden by attribute rules and by default queues use `examples.rules.AttributeRules`. These default rules group cryptors into four strengths:

1. No cryptor
2. XOR cryptor
3. DES cryptor
4. All the other cryptors (triple DES, MARS, RC4, RC6)

Using these rules, a queue accepts messages from a connection if the cryptor on the connection is at least as strong as its own cryptor:

- A queue with no cryptor accepts messages from any connection
- A queue with an XOR cryptor accepts messages from a connection with any cryptor (but not a connection with no cryptor)
- A queue with a DES cryptor accepts messages from a connection with a DES, triple DES, MARS, RC4 or RC6 cryptor
- A queue with a triple DES, MARS, RC4 or RC6 cryptor accepts messages from a connection with any of triple DES, MARS, RC4 or RC6 cryptor

In the previous example, if the Store-and-forward queue used these rules it would not accept a DES encrypted connection because DES is not as strong as its own MARS cryptor, it would throw an ″attribute mismatch″ exception.

A Home-server queue trying to pull messages from a Store-and-forward queue needs a cryptor that is at least as strong as that on the Store-and-forward queue, because the Home-server queue is at the initiating end of the request. Once the Home-server queue has received the message it can store it on a local queue that has any level of protection.

This behavior can be changed by using different attribute rules on the queues. The **equals()** method is used to compare the cryptors, so if the attribute rule has an **equals()** method that always returns `true`, the queue accepts connections with any cryptor.

Trying to send a message from a queue with a weaker cryptor to a queue with a stronger cryptor usually results in an ″attribute mismatch″ exception. However if a connection with a strong cryptor already exists between the queue managers, this can be reused (depending on the attribute rules on the connection) and result in the message being delivered.

Usage scenario

WebSphere MQ Everyplace queue-based security can be used whenever you need to protect the confidentiality of message data being transferred between queue managers.

A typical scenario could be a service that is delivered over an open network, like the internet, where an initiating application makes requests, using a queue manager on a client, 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 WebSphere MQ Everyplace message
2. **putMessage** is used to transfer the message to a queue called `XXX_service_request` on a remote server
3. A queue manager application on the server is setup to listen for messages on the `XXX_service_request` queue
4. When a message event occurs, a **getMessage** is performed, to get the service request message
5. The request is processed (for example by invocation of a CICS transaction on a back-end system)
6. The response (transaction result) is encapsulated in a message
7. **putMessage** is used to return the response to a queue called `XXX_service_reply` on the initiating client queue manager.
8. **waitForMessage** is used on the initiating queue manager to wait for a reply message to arrive in the local queue called `XXX_service_reply`

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 store-and-forward queue that have been directed to the client queue manager. Also, before running any client queue manager application, the *`MQE_ADMIN_ACTION_ADD_QMGR`* option must be used to add a reference to the server queue manager.

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 *MQE_ADMIN_ACTION_ADD_QMGR*, with name ClientQMgr

Secure feature choices

When using queue-based security all the choices for attribute are available:

Authenticator
MQeWTLSCertAuthenticator

Cryptor
MQeXORCryptor or one of the symmetric cryptors MQeDESCryptor, MQe3DESCryptor, MQeRC4Cryptor, MQeRC6Cryptor, or MQeMARSCryptor

Compressor
MQeLZWCompressor, MQeRleCompressor, or MQeGZIPCompressor

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, 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 117, 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.

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 117, 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 128 bit 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 routines 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

To use queue base security, the queue manager that owns the queue must have a private registry. If the MQeWTLSCertAuthenticator is used, the registry must also have its own credentials, which it obtains by auto-registering with the mini-certificate server. In the following example the credentials process is enabled by adding information to the Registry Section of the queue manager's configuration (.ini) file. If the MQeWTLSCertAuthenticator is not used, a private registry is still required but it does not have to register with the mini-certificate server to obtain credentials.

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 117.

Using qm_create to create ClientQMgr and ServerQMgr instances

Note: This example program takes the private registry *PIN*, the *PIN*, the *Certificate-request PIN*, and the *Key Ring Password* from your command line if required. This is convenient for an example but is not recommended for a production system. Care should be taken to prevent the unauthorized disclosure of PINs and passwords.

qm_create 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. If auto-registration with an MiniCertificateServer is required, an WebSphere MQ Everyplace Java MiniCertificateServerGUI must be started, 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 provided on your command line.
2. Use qm_create to create queue managers `ClientQMgr` and `ServerQMgr`. The *PIN*, the *Certificate-request PIN* and the *Key Ring Password* parameters must also be specified if MQeWTLSCertAuthenticator is to be used. For example:

```
qm_create ServerQMgr -p 12345678 -c 12345678 -k It_is_a_secret -s 127.0.0.1:8082

qm_create ClientQMgr -p 12345678 -c 22345678 -k It_is_a_secret -s 127.0.0.1:8082
```

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.

Defining the queues identified for the queue-based scenario described above

There are several ways to 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 then create and use administration messages to add the queue and set the poll interval.

```
q_create ClientQMgr TestClient_HomeServerQ -h ServerQMgr Network:127.0.0.1:8081
TestServer_StoreAndForwardQ -p 12345678 -c 12345678 -k It_is_a_secret
                            -s FastNetwork:127.0.0.1:8082
```

ClientQMgr queues -adding XXX_service_reply queue: Start the `ClientQMgr` locally then create and use an administration messages to add the queue.

```
q_create ClientQMgr XXX_service_reply -r -p 12345678 -c 12345678
-k It_is_a_secret -s FastNetwork:127.0.0.1:8082
```

ServerQMgr queues -adding TestServer_StoreAndForwardQ: Start the `ServerQMgr` locally , create and use an administration messages to add the queue, and then add a remote queue manager reference.

```
q_create ServerQMgr TestServer_StoreAndForwardQ -f ClientQMgr -p 12345678
-c 12345678 -k It_is_a_secret -s FastNetwork:127.0.0.1:8082
```

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.

```
q_create ServerQMgr XXX_service_request -q -p 12345678 -c 12345678
-k It_is_a_secret -s FastNetwork:127.0.0.1:8082
```

Example Server TestService application: An example Server TestService program, securityTestService, is provided. It can be started by the command:

```
securityTestService ServerQMgr ClientQMgr -p 12345678 -c 12345678
-k It_is_a_secret -s FastNetwork:127.0.0.1:8082
```

Client queue manager application initiating XXX_service_request.:

The example queue-based security scenario in "Usage scenario" on page 117 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.

An example Client TestService program, securityTestClient, is also provided. It invokes a service request and processes the resulting reply. This example can be started by the command:

```
securityTestClient ClientQMgr ServerQMgr Network:127.0.0.1:8081 -p 12345678
-c 22345678 -k It_is_a_secret -s FastNetwork:127.0.0.1:8082
```

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 on the command line are:

```
-c 22345678 -k It_is_a_secret -s FastNetwork:127.0.0.1:8082
```

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 describes how to create a queue with name `ServerTestQWTLS` on `ServerQMgr`.

Assume that the mini-certificate server administrator has added `ServerQMgr+ServerTestQWTLS2` with *`Certificate Request PIN`* equal to `12345678`, and has started the mini-certificate server in 'server' mode. Use the `-o` flag on a **q_create** command to create queue `ServerTestQWTLS2` as in the following example:

```
q_create ServerQMgr ServerTestQWTLS2 -o -p 12345678 -c 12345678 -k It_is_a_secret
                                     -s FastNetwork:127.0.0.1:8082
```

Where the 12345678 following -c is ServerTestQWTLS2's *Certificate Request PIN* (not ServerQMgr's). Effectively, the -o flag causes the MQE_QUEUE_TARGETREGISTRY field in the queue creation administration message to be set to MQE_QUEUE_REGISTRYQUEUE, which means that the queue to be created should have its own credentials. (If the queue to be created is sharing its queue manager's credentials, MQE_QUEUE_TARGETREGISTRY must be set to MQE_QUEUE_REGISTRYQMGR. If the queue to be created is not having credentials, MQE_QUEUE_TARGETREGISTRY must be set to MQE_QUEUE_REGISTRYNONE.) The -o also causes the obtained credentials to be published in the queue manager's public registry.

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.

Queue-based security - connection reuse

When data is sent between a queue manager and a remote queue, the queue manager opens a connection to the remote queue manager that owns the queue. By default, if the remote queue is protected, for example with a cryptor, the connection is given exactly the same level of protection as the queue. To reduce the number of connections open concurrently, the queue manager can reuse an existing connection if its level of protection is adequate. If none of the connections has a suitable level of protection, the queue manager can also change the level of protection on an existing connection to match that required for the queue. The default behavior can be changed by using attribute rules on both the queue and the connection. These rules apply to the attribute on the queue (and connection), they are not the same as queue rules.

The C Bindings code API does not support attribute rules written in C. However, users can write their own rules in Java. An example Java rule, examples.rules.AttributeRule, is provided.

While the **examples.rules.AttributeRule** provides practical defaults, there may be a solution specific reason why different behavior is required. You can modify the way connections are reused by extending or replacing the default examples.rules.AttributeRule with rules that define the desired behavior.

If attribute rules are defined for the queue, the queue manager uses the rules to decide whether an existing connection has sufficient protection for the queue. If the **equals()** function in the rules returns true, the connection can be used. WebSphere MQ Everyplace provides an example rule, examples.rules.AttributeRule, that can be used on the queue. This rule allows a connection to be used for a queue if the following conditions are met:

- If the queue has an authenticator, the connection must have the same authenticator. If the queue does not have an authenticator, it does not matter whether the connection has one or not.

- If the queue has a cryptor, the connection must have a cryptor that is the same as or better than that on the queue. If the queue does not have a cryptor it does not matter whether the connection has one or not.
- It does not matter what compressors are defined for the queue or connection

The example rules define ″better″ for a cryptor to mean:

- Any cryptor is the same as or better than XOR
- Any cryptor, except XOR, is the same as or better then DES
- The remaining cryptors (Triple DES, RC4, RC6, and MARS) are considered equal to each other and all better than XOR and DES.

If none of the existing connections has sufficient protection for the queue, the queue manager checks if any of the connections can be upgraded to the required level. If attribute rules are defined for the connection, the **permit()** function is used to determine this. The examples.rules.AttributeRule uses the following criteria:

- If the connection has been authenticated it cannot be upgraded, but if it does not have one, an authenticator can be added to a connection.
- A cryptor can be added to a connection or strengthened (using the criteria for ″better″ described above). A cryptor cannot be removed from the connection or replaced with a weaker cryptor.
- A compressor can be changed, added to, or removed from the connection.

Before allowing connection reuse, the target queue uses its current AttributeRule **equals()** function to determine if the connection attribute can provide an appropriate level of protection for the target queue. This provides protection against inconsistency in the queue attribute rules on the local and target queue managers.

Attribute rules are set on a queue when it is created or modified using administration messages. Attribute rules are set on connections by defining an alias on *ChannelAttrRules*. For example, the following pseudo-code shows how to make a queue manager use examples.rules.AttributeRule.

```
mqe_alias(pErrorBlock, MQeString("ChannelAttrRules"),
          MQeString("examples.rules.AttributeRule"));
```

It is possible to run without setting ChannelAttrRules, but this mode of operation is not recommended.

Message-level security

Message-level security facilitates the protection of message data between an initiating and receiving WebSphere MQ Everyplace application. Message-level security is an application layer service. It requires the initiating WebSphere MQ 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 function, which invokes the (attached) attribute's **encodeData()** function to protect the message data. The receiving application's queue manager handles the application's

getMessage() with the message's 'restore' function which in turn uses the supplied attribute's **decodeData()** function 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 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.

WebSphere MQ 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

WebSphere MQ 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 WebSphere MQ Everyplace symmetric cryptor and compressor choices. Like local security it requires the attribute's key to be preset before it is supplied as a parameter on **putMessage()** and **getMessage()**. This provides a simple and powerful method for message-level protection that enables use of strong encryption to protect message confidentiality, without the overhead of any public key infrastructure (PKI).

MQeMTrustAttribute
: This provides a more advanced solution using digital signatures and exploiting the default public key infrastructure to provide a digital envelope style of protection. It uses ISO9796 digital signature/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 WebSphere MQ Everyplace symmetric cryptor and compressor choices. Chosen for size optimization, the certificates used are mini-certificates which conform to the WTLS Specification approved by the WAP forum. WebSphere MQ Everyplace provides a default public key infrastructure to distribute the certificates as required to encrypt and authenticate the messages.

A typical MQeMTrustAtribute protected message has the format:

```
RSA-enc{SymKey}, SymKey-enc{Data, DataDigest, DataSignature}
```

where:

RSA-enc:
RSA encrypted with the intended recipient's public key, from his mini-certificate

SymKey:
Generated pseudo-random symmetric key

SymKey-enc:
Symmetrically encrypted with the *SymKey*

Data: Message data

DataDigest:
Digest of message data

DigSignature:
Initiator's digital signature of message data

Selection Criteria

MQeMAttribute relies totally on the solution owner to manage the content of the key seed that is used to derive the symmetric key 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 WebSphere MQ 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 decide that MQeMAttrribute suits their needs, while solutions for which one to one (authenticatable entity to authenticatable entity) transfer and non-repudiation of the message originator are important may find MQeMTrustAttribute is the correct choice.

Usage guide

The following pseudo-code fragments provide examples of how to protect and unprotect a message using MQeMAttribute and MQeMTrustAttribute

WebSphere MQ Everyplace message-level security using MAttribute

```
/*SIMPLE PROTECT FRAGMENT */

      MQeMsgHndl msgObj;
      MQeMAttributeHndl attr = null;
      MQeInt64 confirmId;
  MQERETURN rc;
  MQeExceptBlock exceptBlock;
  MQe3DESCryptorHndl tdes;
  MQeMAttributeHndl attr;
```

```
MQeKeyHndl localkey;
       MQeMsgHndl msgObj;

     /* create the cryptor */
rc = mqe_uniqueValue(&exceptBlock, &confirmId);
       rc = mqe3DESCryptor_new(&exceptBlock, &tdes);
        /* create an attribute using the cryptor */
       rc = mqeMAttribute_new(&exceptBlock,&attr, NULL,tdes,NULL);
        /* create a local key */
       rc = mqeKey_new(&exceptBlock,&localkey);
        /* give it the key seed */
       rc = mqeKey_setLocalKey(localkey, new(&exceptBlock,
                               MQeString("my secret key"));
        /* set the key in the attribute */
       rc = mqeMAttribute_setKey(attr, &exceptBlock, localkey );
        /* create the message */
       rc = mqeMsg_new(&exceptBlock, &msObj);
       rc = mqeFields_putAscii(msgObj, &exceptBlock, MQeString("MsgData"),
                               MQeString("0123456789abcdef..."));
        /* put the message using the attribute */
       rc = mqeQueueManager_putMessage(newQM, &exceptBlock, targetQMgrName,
                                       targetQName, msgObj, attr, confirmId );


    /*SIMPLE UNPROTECT FRAGMENT */

      MQeMsgHndl msgObj2;
      MQeMAttributeHndl attr2;
      MQeInt64 confirmId2;
MQeKeyHndl localkey;
MQERETURN rc;
MQeExceptBlock exceptBlock;

rc = mqe_uniqueValue(&exceptBlock, &confirmId2);
     /* create the attribute - we do not have to specify the cryptor, */
       /* the attribute can get this from the message itself  */
      rc = mqeMAttribute_new(&exceptBlock, &attr2, NULL,
         NULL, NULL);
       /* create a local key */
      rc = mqeKey_new(&exceptBlock, &localkey);
       /* give it the key seed */
      rc = mqeKey_setLocalKey(localkey,
         &exceptBlock,MQeString("my secret key"));
       /* set the key in the attribute */
      rc = mqeMAttribute_setKey(attr2, &exceptBlock, localkey );
       /* get the message using the attribute */
      rc = mqeQueueManager_getMessage(newQM, &exceptBlock, &msgObj2,
             targetQMgrName, targetQName, NULL,
               attr2, confirmId2 );
```

WebSphere MQ Everyplace message-level security using MTustAttribute

```
/*SIMPLE PROTECT FRAGMENT */

    MQeMsgHndl msgObj;
    MQeMTrustAttributeHndl attr;
MQeMARSCryptorHndl mars;
MQePrivateRegistryHndl sendreg;
MQePublicRegistryHndl pr;
MQeMsgObjectHndl msgObj;
    MQeInt64 confirmId;
MQERETURN rc;
MQeExceptBlock exceptBlock;

     rc = mqe_uniqueValue(&exceptBlock, &confirmId);
/* create the cryptor */
```

```
    rc = mqeMARSCryptor_new(&exceptBLock, &mars);
     /* create an attribute using the cryptor */
    rc = mqeMTrustAttribute_new(&exceptBLock, &attr, NULL,
           mars, NULL);
     /* open the private registry belonging to the sender */
    rc = mqePrivateRegistry_new(&exceptBLock, &sendreg);
    rc = mqePrivateRegistry_activate(sendreg, &exceptBLock,
             MQeStrinh("Bruce1"),
                                       MQeString(".//MQeNode_PrivateRegistry"),
                                       MQeString("12345678"),
                                       MQeString("It_is_a_secret"), NULL, NULL);
     /* set the private registry in the attribute */
    rc = mqeMTrustAttribute_setPrivateRegistry(attr, &exceptBLock, sendreg);
     /* set the target (recipient) name in the attribute */
    rc = mqeMTrustAttribute_setTarget(attr, &exceptBLock,
             MQeString("Bruce8"));
     /* open a public registry to get the target's certificate */
    rc = mqePublicRegistry_new(&exceptBLock, &pr);
       rc = mqePublicRegistry_activate(pr, &exceptBLock,
                                       MQeString("MQeNode_PublicRegistry"),
                                       MQeString(".//"));
     /* set the public registry in the attribute */
    rc = mqeMTrustAttribute_setPublicRegistry(attr, &exceptBLock, pr);
     /* set a home server, which is used to find the certificate*/
     /* if it is not already in the public registry */
    rc = mqeMTrustAttribute_setHomeServer(attr, &exceptBLock,
                                          MQeString(MyHomeServer ":8082"));
     /* create the message */
    rc = mqeMsgObject_new(&exceptBLock, &msObj);
    rc = mqeFields_putAscii(msgObj, &exceptBLock, MQeString("MsgData"),
                            MQeString("0123456789abcdef..."));
     /* put the message using the attribute */
    rc = mqeQueueManager_putMessage(newQM, &exceptBLock, targetQMgrName,
             targetQName, msgObj,
             attr, confirmId );

    /*SIMPLE UNPROTECT FRAGMENT */

    MQeMsgHNDL msgObj2;
    MQeMTrustAttributeHndl attr2;
MQeMARSCryptorHndl mars;
MQePrivateRegistryHndl getreg;
 MQePublicRegistryHndl pr;
   MQeInt64 confirmId2;
MQERETURN rc;
MQeExceptBlock exceptBlock;

rc = mqe_uniqueValue(&exceptBLock, &confirmId);
     /* create the cryptor */
    rc = mqeMARSCryptor_new(&exceptBLock, &mars);
     /* create an attribute using the cryptor */
    rc = mqeMTrustAttribute_new(&exceptBLock,&attr2, NULL,
           mars, NULL);
     /* open the private registry belonging to the target */
    rc = mqePrivateRegistry_new(&exceptBLock, &getreg);
    rc = mqePrivateRegistry_activate(getreg, &exceptBLock,
              MQeStrinh("Bruce8"),
                                       MQeString(".//MQeNode_PrivateRegistry"),
                                       MQeString("12345678"),
                                       MQeString("It_is_a_secret"), NULL, NULL);
     /* set the private registry in the attribute */
    rc = mqeMTrustAttribute_setPrivateRegistry(attr2, &exceptBLock, getreg);
     /* open a public registry to get the sender's certificate */
    rc = mqePublicRegistry_new(&exceptBLock, &pr);
    rc = mqePublicRegistry_activate(pr, &exceptBLock,
                                       MQeString("MQeNode_PublicRegistry"),
                                       MQeString(".//"));
```

```
      /* set the public registry in the attribute */
     rc = mqeMTrustAttribute_setPublicRegistry(attr2, &exceptBLock, pr);
      /* set a home server, which is used to find the certificate*/
      /* if it is not already in the public registry */
     rc = mqeMTrustAttribute_setHomeServer(attr2, &exceptBLock,
                                           MQeString(MyHomeServer ":8082"));
      /* get the message using the attribute */
 rc = mqeQueueManager_getMessage(newQM, &exceptBLock, &msgObj2,
             targetQMgrName, targetQName, NULL,
             attr2, confirmId2 );
```

Non-repudiation

The MQeMTrustAttribute digitally signs the message. This enables the recipient to validate the creator of the message, and ensures that the creator cannot later deny creating the message. This is known as *non-repudiation*. This process depends on the fact that only one public key (certificate) can validate the signature successfully, and this proves that the signature was created with the corresponding private key. The only way the alleged creator can deny creating the message is to claim that someone else had access to the private key.

When a message is created with the MQeMTrustAttribute, it uses the private key from the sender's private registry to create the digital signature and it stores the sender's name in the message. When the message is read (with the queue manager's **getMessage()** function), it uses the sender's public certificate to validate the digital signature. The message is read successfully only if the signature validates successfully, proving that the message was created by the entity whose name was stored in the message as the sender.

When the MQeMTrustAttribute is specified as a parameter to the queue manager's **getMessage()** function, the attribute validates the digital signature but by the time the message is returned to the user's application all the information relating to the signature has been discarded. If non-repudiation is important to you, you must keep a record of this information. The simplest way to do this is to keep a copy of the encrypted message, because that includes the digital signature. You can do this by using the **getMessage()** function without an attribute. This returns the encrypted message which you can then save, for example in a local queue. You can decrypt the message by applying the attribute to access the contents of the message.

The following pseudo-code fragment provides an example of how to save an encrypted message.

Saving a copy of an encrypted message

```
/*SIMPLE FRAGMENT TO SAVE ENCRYPTED MESSAGE*/

  MQeMsgHndl msgObj2, tmpMsg1, tmpMsg2;
  MQeMTrustAttributeHndl attr2;
  MQeMARSCryptorHndl mars;
  MQePrivateRegistryHndl getreg;
   MQePublicRegistryHndl pr;
  MQeInt64 confirmId2, confirmId3;
  MQERETURN rc;
  MQeExceptBlock exceptBlock;

  rc = mqe_uniqueValue(&exceptBlock, &confirmId2);
  rc = mqe_uniqueValue(&exceptBlock, &confirmId3);

      /* read the encrypted message without an attribute */
      rc = mqeQueueManager_getMessage(newQM, &exceptBlock,
              &tmpMsg1,
```

```
                                    targetQMgrName,targetQName,
                                    NULL, NULL, confirmId2 );
/* save the encrypted message - we cannot put it directly */
/* to another queue because of the origin queue manager  */
/* data. Embed it in another message     */
rc = mqeMsg_new(&exceptBlock, &tmpMsg2);
rc = mqeFields_putFields(tmpMsg2, &exceptBlock,
                           MQeTring("encryptedMsg"), tmpMsg1);
rc = mqeQueueManager_putMessage(newQM, &exceptBlock, localQMgrName,
                                    archiveQName, tmpMsg2, NULL, confirmId3);
/* now decrypt and read the message ... */
/* create the cryptor */
rc = mqeMARSCryptor_new(&exceptBlock, &mars);
/* create an attribute using the cryptor */
rc = mqeMTrustAttribute_new(&exceptBlock, &attr2,
       NULL, mars, NULL);
/* open the private registry belonging to the target */
rc = mqePrivateRegistry_new(&exceptBlock, &getreg);
rc = mqePrivateRegistry_activate(getreg, &exceptBlock,
        MQeStrinh("Bruce8"),
                                   MQeString(".//MQeNode_PrivateRegistry"),
              MQeString("12345678"), MQeString("It_is_a_secret"),
             NULL, NULL);
/* set the private registry in the attribute */
rc = mqeMTrustAttribute_setPrivateRegistry(attr2, &exceptBlock,
           getreg);
/* open a public registry to get the sender's certificate */
rc = mqePublicRegistry_new(&exceptBlock, &pr);
rc = mqePublicRegistry_activate(pr, &exceptBlock,
                                    MQeString("MQeNode_PublicRegistry"),
                                    MQeString(".//"));
/* set the public registry in the attribute */
 rc = mqeMTrustAttribute_setPublicRegistry(attr2, &exceptBlock, pr);
/* set a home server, which is used to find the certificate*/
/* if it is not already in the public registry */
rc = mqeMTrustAttribute_setHomeServer(attr2, &exceptBlock,
                                        MQeString(MyHomeServer ":8082"));
/* decrypt the message by unwrapping it */
rc = mqeMsg_unwrapMsgObject(tmpMsg1, &exceptBlock,
       msObj2, attr2);
```

Private registry service

This section describes the private registry service provided by WebSphere MQ 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.

WebSphere MQ 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 application selected mapping.

Internally, WebSphere MQ Everyplace defines all queue managers that can either originate or be the target of mini-certificate dependent services as authenticatable entities. WebSphere MQ Everyplace also defines queues defined to use

mini-certificate based authenticators as authenticatable 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.

WebSphere MQ Everyplace provides configurable options to enable queue managers and queues to auto-register as an authenticatable entity. WebSphere MQ Everyplace private registry service (MQePrivateRegistry) provides services that enable an WebSphere MQ 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 authenticatable entity needs its own credentials. This provides two challenges, firstly how to execute registration to get the credentials, and secondly where to manage the credentials in a secure manner. WebSphere MQ 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

WebSphere MQ 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 WebSphere MQ Everyplace application uses MQePrivateRegistry directly to create a new authenticatable entity. When registration is triggered, new credentials are created and stored in the authenticatable entity's private registry. Auto-registration steps include generating a new RSA key pair, protecting and saving the private key in the private registry; and packaging the public key in a new-certificate request to the default mini-certificate server. Assuming the mini-certificate server is configured and available, and the authenticatable entity has been pre-registered by the mini-certificate server (is authorized to have a certificate), the mini-certificate server returns the authenticatable entity's new mini-certificate, along with its own mini-certificate 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 public key (mini-certificate).

Usage scenario

The primary purpose of WebSphere MQ Everyplace's private registry is to provide a private repository for WebSphere MQ 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 WebSphere MQ Everyplace security features:

Queue-based security with MQeWTLSCertAuthenticator
Whenever queue-based security is used, where a queue attribute is defined with MQeWTLSCertAuthenticator, (mini-certificate based mutual authentication) the authenticatable entities involved are WebSphere MQ 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 WebSphere MQ 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 WebSphere MQ 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
WebSphere MQ 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, WebSphere MQ Everyplace owned authenticatable entities must have credentials. This is achieved by completing the correct configuration so that auto-registration of queue managers is triggered. This requires the following steps:

1. Setup and start an instance of WebSphere MQ 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. Refer to the description in "Using qm_create to create ClientQMgr and ServerQMgr instances" on page 119.

Prior to using message-level security to protect messages using MQeMTrustAttribute, the application must use private registry services to ensure that the initiating and recipient entities have credentials. This requires the following steps:

1. Setup and start an instance of WebSphere MQ 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 code fragment below to trigger auto-registration of the application entity . This creates the entity's credentials and saves them in its private registry.

```
/* SIMPLE MQePrivateRegistry FRAGMENT                                  */
  MQePrivateRegistryHndl preg;
  MQERETURN rc;
  MQeExceptBlock exceptBlock;

       /* setup PrivateRegistry parameters                               */
        rc = mqePrivateRegistry_new(&Block, &preg);
        rc = mqePrivateRegistry_activate(
   preg,
   &Block,
   MQeString("Bruce"),     /* entity name  */
              MQeString(".//MQeNode_PrivateRegistry"),
     /* directory root  */

     MQeString("11111111"),
           /* private reg access PIN     */
               MQeString("It_is_a_secret"),
    /* private credential keyseed */
               MQeString("12345678"),
          /* on-time-use Cert Req PIN   */
              MQeString("9.20.X.YYY:8082"));
     /* addr and port MiniCertSvr  */
```

Public registry service

This section describes the public registry service provided by WebSphere MQ Everyplace.

WebSphere MQ Everyplace provides default services facilitating the sharing of authenticatable entity *public credentials* (mini-certificates) between WebSphere MQ Everyplace nodes. Access to these mini-certificates is a prerequisite for message-level security. WebSphere MQ 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. WebSphere MQ 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 WebSphere MQ 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 WebSphere MQ 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 WebSphere MQ Everyplace client to automatically get the mini-certificates of other authenticatable entities that it needs, from its WebSphere MQ Everyplace home server, and then save them in its public registry.

Secure feature choices

It is the Solution creator's choice whether to use the public registry active features for sharing and getting mini-certificates between the public registries of different WebSphere MQ Everyplace nodes.

The alternative to this intelligent replication may be to have an out-of-band utility to initialize an WebSphere MQ 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 WebSphere MQ 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 WebSphere MQ 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 WebSphere MQ Everyplace node public registry.

Usage guide

The following code segment demonstrates how to share certificates among a group of queue managers:

```
/*SIMPLE MQePublicRegistry shareCertificate FRAGMENT */
 MQePublicRegistryHndl pubreg;
 MQePrivateRegistryHndl preg;
 MQERETURN rc;
 MQeExceptBlock exceptBlock;
 MQeStringHndl hEntityName;
 MQeFieldsHndl hCert;
 MQEINT32 i;

 /*instantiate and activate PublicReg */
 rc = mqePublicRegistry_new(&exceptBlock, &pubreg);
 rc = mqePublicRegistry_activate(pubreg, &exceptBlock,
                                    MQeString("MQeNode_PublicRegistry"),
            MQeString(".\\"));
 /* auto-register Bruce1,Bruce2...Bruce8 */
 /* ... note that the mini-certificate issuance service must */
 /* have been configured to allow the auto-registration    */
 for (i = 1; i < 9; i++)
 {
  rc = MQeString_new(&exceptBlock,
          &hEntityName, strcat("Bruce" + itoa(i)));
  rc = mqePrivateRegistry_new(&exceptBlock, &preg);
  /* activate() will initiate auto-registration */
  rc = mqePrivateRegistry_activate(
   preg,
   &exceptBlock,
   hEntityName,
   MQeString(".\\MQeNode_PrivateRegistry"),
    MQeString("12345678"),
   MQeString("It_is_a_secret"),
   MQeString("12345678"),
   MQeString("9.20.X.YYY:8082")
       );
  /* save MiniCert from PrivReg in PubReg*/
```

```
 rc = mqePrivateRegistry_getCertificate(preg, &exceptBlock,
              &hCert, hEntityName);
 rc = mqePublicRegistry_putCertificate(pubreg, &exceptBlock,
              hEntityName, hCert);
 /* before share of MiniCert */
 rc = mqePublicRegistry_shareCertificate(pubreg,&exceptBlock,
              hEntityName, hCert,
              MQeString("9.20.X.YYY:8082"));
 rc = mqePrivateRegistry_close(preg, &exceptBlock);
 (void)mqePrivateRegistry_free(preg, NULL);
 (void)mqeString_free(hEntityName, NULL);
 (void)mqeFields_free(hCert, NULL);
}
rc = mqePublicRegistry_close(pubreg, &exceptBlock);
(void)mqePublicRegistry_close(pubreg, NULL);
```

Notes:

1. It is not possible to activate a registry instance more than once, hence the example above demonstrates the recommended practice of accessing a private registry by creating a new instance of MQePrivateRegistry, activating the instance, performing the required operations and closing the instance.
2. If you want to share certificates using a public registry on the home-server, the public registry must be called `MQeNode_PublicRegistry`.

mini-certificate issuance service

Please refer to the corresponding section in *WebSphere MQ Everyplace Application Programming Guide*

Appendix A. Applying maintenance to WebSphere MQ Everyplace

Maintenance updates for WebSphere MQ Everyplace are always shipped as a complete new release. There are two options when upgrading from one release to another:

Completely uninstall the current level, and install the new level in same directory
When doing this it is recommended you keep the install package for the current level to allow it to be restored later if necessary.

Keep the existing level and install the new level into a new directory
After installation, check your classpath to ensure that the latest level of WebSphere MQ Everyplace is being invoked. If installing on Windows, make sure that you give the shortcuts folder for the new install a different name to the existing one.

For more general information on maintenance updates and their availability see the WebSphere MQ family Web page at *http://www.software.ibm.com/mqseries/*.

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,

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 Everyplace IBM iSeries MQSeries WebSphere z/OS zSeries

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 the 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 WebSphere MQ Everyplace object that allows messages to flow between WebSphere MQ Everyplace and other messaging systems, including WebSphere MQ.

channel. See *dynamic channel*, client/server channel, peer channel, and *MQI channel*.

channel manager. An WebSphere MQ 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. (1)In WebSphere MQ Everyplace, a client is WebSphere MQ Everyplace code running without a channel manager or channel listener. Contrast with *server (1)*. (2)In WebSphere MQ, a client is a run-time component that provides access to queuing services on a server for local user applications.

client/server channel. An WebSphere MQ Everyplace a unidirectional channel between a client and a server that can only be established from the client side. Contrast with *peer channel*.

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.

device. A small portable machine running WebSphere MQ Everyplace as a client. Contrast with *server(1)*.

dynamic channel. This is a name given to WebSphere MQ Everyplace channels that connect clients and servers to enable the transfer of messages. They are called *dynamic* because they are created on demand. See *client/server* and *peer* channels. Contrast with*MQI channel*.

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 function calls.

gateway. An WebSphere MQ Everyplace gateway is a computer running the WebSphere MQ Everyplace WebSphere MQ bridge code.

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 functions and no instance variables. An interface provides a common set of functions 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

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 a WebSphere MQ client to a queue manager on a server system and transfers MQI calls and responses in a bidirectional manner. MQI channels must be explicitly created. Contrast with *dynamic channels*.

WebSphere MQ. WebSphere MQ 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 WebSphere MQ, 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 functions and fields in the classes.

peer channel. A bidirectional WebSphere MQ Everyplace channel, normally used between clients. The connection can be established from either end.

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 function or variable is visible everywhere that its class is visible

queue. A queue is a WebSphere MQ 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 WebSphere MQ Everyplace server is WebSphere MQ Everyplace code with an WebSphere MQ Everyplace channel manager, and WebSphere MQ Everyplace channel listener, configured. This provides the ability to receive from multiple devices and servers concurrently. Contrast with *client (1)*. (2)A computer running WebSphere MQ Everyplace server code. Contrast with *device*. (3) A WebSphere MQ server is a queue manager that provides message queuing services to client applications running on a remote workstation. (4) More generally, a server is a program that responds to requests for information in the particular two-program information flow model of client/server, or 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 functions and variables of its superclass.

superclass. A superclass is a class that is extended by some other class. The superclass's public and protected functions 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:

- *WebSphere MQ Everyplace Read Me First*, SC34-6276-01
- *WebSphere MQ Everyplace Introduction*, SC34-6277-01
- *WebSphere MQ Everyplace Programming Reference*, SC34-6279-01
- *WebSphere MQ Everyplace Application Programming Guide*, SC34-6278-01
- *WebSphere MQ Everyplace C Programming Reference*
- *WebSphere MQ Everyplace C Programming Guide for Palm OS*, SC34-6281-01
- *WebSphere MQ An Introduction to Messaging and Queuing*, GC33-0805-01

Index

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 make comments about the functions of IBM products or systems, 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
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–842327
  - From within the U.K., use 01962–842327
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Printed in U.S.A.

SC34-6280-01