MQSeries® Everyplace for Multiplatforms

Introduction

Version 1.2

GC34-5843-02

MQSeries® Everyplace for Multiplatforms

Introduction

Version 1.2

GC34-5843-02

Take Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix. Notices" on page 81

Licence warning

MQSeries Everyplace Version 1.2 is a toolkit that enables users to write MQSeries 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 licences.

To use MQSeries Everyplace on specified server platforms (other than for purposes of code development and test), *capacity-unit use authorizations* (which are recorded on Proof of Entitlement documents and valid to support use of MQSeries Everyplace according to published capacity unit and pricing group tables) must be obtained in order to be licensed to use the program on each machine and machine upgrade.

Device platform use authorizations (which are recorded on Proof of Entitlement documents and valid to support use of MQSeries Everyplace) are required to use the product (other than for purposes of code development and test) on specified client platforms. These licenses do not entitle the user to use the MQSeries Everyplace Bridge, or to run on the server platforms specified in the MQSeries Everyplace pricing group lists published by IBM and also available on the Web via the URL mentioned below:

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

Third Edition (May 2001)

This edition applies to MQSeries Everyplace Version 1.2 and to all subsequent releases and modifications until otherwise indicated in new editions.

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

Contents

About this book

This book is a general introduction to MQSeries Everyplace for Multiplatforms product (generally referred to in this book as MQSeries Everyplace). It covers the product concepts and its relationship to other MQSeries products.

For detailed information on the MQSeries Everyplace API and how to use it to create MQSeries Everyplace applications, see the *MQSeries Everyplace for Multiplatforms Programming Reference* and the *MQSeries Everyplace for Multiplatforms Programming Guide*.

For information relating to using other programming languages with MQSeries Everyplace for Multiplatforms see *MQSeries Everyplace for Multiplatforms Native Client Information*

For MQSeries Everyplace for Multiplatforms installation procedures see *MQSeries Everyplace for Multiplatforms Read Me First*

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

Who should read this book

This book is intended for those interested in using secure messaging on lightweight devices such as sensors, phones, Personal Digital Assistants (PDAs) and laptop computers, and those with a need to extend the scope of an MQSeries Everyplace messaging network.

Prerequisite knowledge

No previous knowledge is required to read this information, but an initial understanding of the concepts of secure messaging is an advantage.

If you do not have this understanding, you may find it useful to read the following MQSeries book:

- *MQSeries An Introduction to Messaging and Queuing*

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

Terms used in this book

The following terms are used throughout this book:

MQSeries family
refers to the following MQSeries products:

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

MQSeries Messaging
refers to the following messaging product groups:

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

MQSeries
refers to the following three MQSeries Messaging product groups:

- Distributed messaging
- Host messaging
- Workstation messaging

MQSeries Everyplace
Refers to the third MQSeries Messaging product group, pervasive messaging.

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

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

Gateway
A computer of any size running MQSeries Everyplace programs that include MQSeries-bridge function.

Summary of Changes

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

Changes for this edition (GC34-5843-02)

Much of the text has been restructured and rewritten, and the following information has been added:

- Additional platforms supported
- Getting started information

Changes for previous edition (GC34-5843-01)

The following information has been added:

- Information for using MQSeries Everyplace on AIX and Solaris.
- Storage requirements.
- Readers comment form.

Chapter 1. Overview

MQSeries Everyplace is a member of the MQSeries family of business messaging products. It is designed to satisfy the messaging needs of lightweight devices, such as sensors, phones, Personal Digital assistants (PDAs) and laptop computers, as well as supporting mobility and the requirements that arise from the use of fragile communication networks. It maintains the standard MQSeries quality of service, providing once-only assured delivery, and exchanges messages with other family members. Since many MQSeries Everyplace applications run outside the protection of an Internet firewall, it also provides sophisticated security capabilities.

Lightweight devices require the messaging subsystem to be frugal in its use of system resources and consequently MQSeries Everyplace offers tailored function and interfaces appropriate to its customer set and does not aim to provide exactly the same capabilities as other members of the family. On the other hand, it does include unique function in order to support its particular classes of user, such as comprehensive security provision, message objects, synchronous and asynchronous messaging, remote queue access, and message push and pull.

MQSeries Everyplace is also designed to integrate well with other members of the IBM pervasive computing family and other components of the Websphere Everyplace Server.

Chapter 2. Software environments

Supported platforms

MQSeries Everyplace is only directly installable on certain server platforms. To transfer programs and Java classes to other platforms, an appropriate download or file transfer program (not supplied) must be used.

Directly supported platforms with installation support

The following platforms are those on which the product can be installed using the built-in tools.

- Windows NT® v4
- Windows® 2000
- Windows 95/98/ME
- AIX Version 4.3
- Sun Solaris Version 7 or 8
- Linux Intel Kernel 2.2 (installed using a zip file).
- HP-UX 11.0 (installed using a zip file)

Directly supported platforms without installation support

The following platforms are supported for the testing and deployment of MQSeries Everyplace, but only support installation by file transfer from another platform.

- WinCE 2.1 running on HP Jornada devices (Models 680 or 820)
- EPOC 32 bit Release 5running on Psion devices (5MX Pro or NetBook)
- PalmOS, V3.0 or higher running on Palm V and IBM Workpad C3
- IBM 4690 OS with Java

Indirectly supported platforms

The following platforms may be used, but are only supported if their Java environment is fully compatible with that on the directly supported platforms. Problems can only be investigated on one of the tested platforms listed above.

- Linux on zSeries running Kernel 2.2
- iSeries
- OS/2
- EPOC (on devices other than those listed above)
- WinCE (on devices other than those listed above)

- QNX Neutrino
- Pocket PC
- PalmOS (on devices other than those listed above)
- Any other platform running one of the Java environments listed in "Java environment"

Java environment

One of the following Java runtime environments is required:

- IBM Java runtime (JVM 1.3 or later), including Java Micro Edition
- Any Java which is Sun Java (V1.1 or later) certified [1]

Note: The Java needs to be fully compatible with that tested on one of the following platforms in order for service to be available

- HP Jornada devices (Models 680 or 820) running Windows CE operating system
- Psion devices (5MX Pro or NetBook) running EPOC operating system
- One of the server platforms in the directly supported lists above

MQSeries Classes for Java is required for MQSeries-bridge operation. You should check the level of Java that is required to run the version of MQSeries Classes for Java.

Personal Java

Personal Java may be used instead of other JVMs on device platforms.

To use MQSeries Everyplace the following optional classes of Personal Java are required:

- For MQSeries Everyplace base classes:
  - java.io.FileInputStream
  - java.io.FileOutputStream
  - java.io.File
  - java.io.FilenameFilter
- To use the MQeGZIPCompressor:
  - java.util.zip.GZIPOutputStream
- To use any encryption
  - java.math.BigInteger

1. You may experience problems if you run the installer under Sun's JVM with the JIT (Just In Time) compiler enabled. If you use a Sun JVM we recommend that you disable the JIT compiler using the command: `java -Djava.compiler=NONE install`

The MQSeries Everyplace examples require some of the optional classes in packages java.io and java.awt.

Storage requirements

The following table shows the storage you need to perform the installation of MQSeries Everyplace.

Table 1. Storage required to perform installation

Operating system	Storage required
Windows NT (file system = NTFS)	26Mb
AIX	29Mb
Solaris	27Mb

The following table shows the storage you need for the MQSeries Everyplace files after installation.

Table 2. Storage required for MQSeries Everyplace

Operating system	Storage required
Windows NT (file system = NTFS)	9.5Mb
AIX	11Mb
Solaris	10Mb

Chapter 3. The MQSeries family

The MQSeries family includes many products, offering a range of capabilities, as illustrated in Figure 1

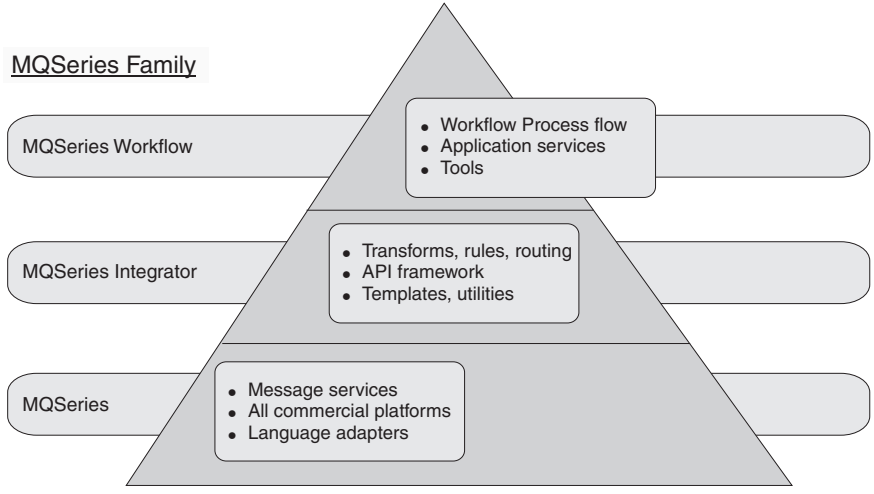


Figure 1. The MQSeries family

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

Both MQSeries Workflow and MQSeries Integrator products take advantage of the connectivity provided by the MQSeries messaging layer.

MQSeries family messaging is supplied by both MQSeries and MQSeries Everyplace products; each being designed to support one or more hardware server platforms and/or associated operating systems. Given the wide variety in platform capabilities, these individual products are organized into product groups, reflecting common function and design. Three such product groups exist:

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

Messaging itself, irrespective of the particular product or product group, is based on queue managers. Queue managers manage queues that can each store messages. Applications communicate with a local queue manager, and get or put messages to queues. If a message is put to a remote queue, that is one owned by a remote queue manager, the message is transmitted over channels to the remote queue manager. In this way messages can hop through one or more intermediate queue managers before reaching their destination. The essence of messaging is to uncouple the sending application from the receiving application, queuing messages at intermediate points if necessary. All MQSeries messaging products are concerned with the same basic elements of queue managers, queues, messages and channels, though there are many differences in detail.

MQSeries host and distributed products

MQSeries host and distributed messaging products are used to support many different network configurations, all of which involve clients and servers,[2] some examples of which are illustrated in Figure 2.

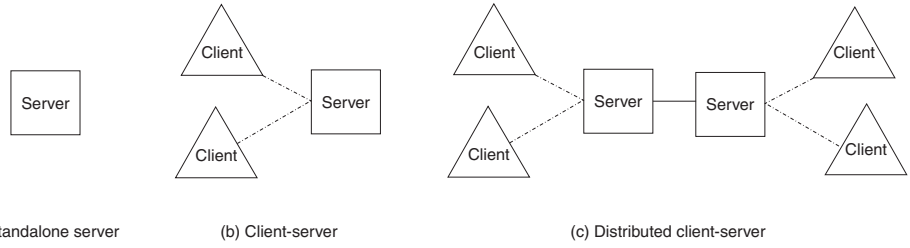


Figure 2. Simple host and distributed configurations

In the simplest case a standalone server is configured, running a queue manager. One or more applications run on that server, exchanging messages via queues. An alternative configuration is client-server. Here the queue manager only exists on the server, but the clients each have access to it through a *client channel*. The client channel is a bidirectional communications link that flows a unique MQSeries protocol implementing something similar to a remote procedure call (RPC). Applications can run on the clients, accessing server queues. One advantage of the client-server configuration is

2. Note that these terms have very specific meanings within MQSeries host, distributed, and workstation messaging products, that do not always correspond to their more common usage.

that the client-messaging infrastructure is lightweight, being dependent on the server queue manager. A disadvantage is that clients and their associated server operate synchronously and therefore require the client channel to be always available.

The distributed client-server configuration shows a more complex case, with multiple servers involved. In these configurations servers exchange messages through *message channels*. Message channels are unidirectional, with a protocol designed for the safe, asynchronous exchange of message data. These message channels need not be available for the clients to continue processing, though no messages can flow between servers when communications are not available.

MQSeries Everyplace

MQSeries Everyplace supports a wide variety of network configurations through the provision of queue managers, with each queue manager enabled with appropriate capabilities. There is no concept of a client or a server as in the MQSeries host or distributed products. MQSeries Everyplace queue managers can act as traditional clients or servers but each is in fact simply a queue manager enabled to perform application-defined tasks. As an illustration of this an MQSeries Everyplace queue manager can be configured with or without local queues. With local queues it can store messages locally and hence offer applications asynchronous messaging, but without local queues it is restricted to synchronous messaging. Another example of tailored configuration is that a queue manager may be configured with or without bridge capabilities. With the bridge it has the ability to exchange messages with MQSeries host or workstation queue managers; without the bridge it can only communicate directly with other MQSeries Everyplace queue managers (although it can communicate indirectly through other queue managers in the network that have bridge capabilities).

Note: A new node for MQSeries Integrator (MQSI) allows connection of MQSeries Everyplace directly, without the use of the MQSeries-bridge.

MQSeries Everyplace queue managers use *dynamic channels* to exchange information. These dynamic channels have different properties from the *client channels* and *message channels* used by other members of the family. Some of the key features of dynamic channels are:

- Support for both *synchronous* and *asynchronous* messaging: Synchronous messaging provides a transmission service directly from the source application to the target queue, without queuing at the source queue manager. Asynchronous messaging is a transmission service from the source queue manager to the target queue, with possible queuing at the source queue manager.

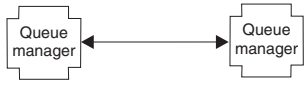
- *End-to-end service* provision: Channels go from the source queue manager to a destination queue manager, possibly running through intermediate queue managers. The underlying transport protocol used can change as the channel passes through these intermediates.
- Support for *compression*, *encryption*, and *authentication*: Channels have these security characteristics to protect the data in transit.
- Support for *peer-to-peer* operation and *client-server* operation: Channels defined as peer-to-peer are functionally symmetrical, such that the either the source or the target can initiate an operation over the channel. Client-server channels are request/response, the client makes a request of the server and the server responds to that request. (Note that this does not restrict the message flow. Messages can flow from client to server and from server to client).

The bridge configuration option allows an MQSeries Everyplace queue manager to communicate with MQSeries host and distributed queue managers through *client channels*. The bridge component manages a pool of client channels that can be attached to one or more host or distributed queue managers. Multiple bridge-enabled MQSeries Everyplace queue managers can be configured in a single network to provide the required capacity, performance and reliability.

Some typical, but arbitrary, MQSeries Everyplace configurations are shown in the following diagrams. For clarity the diagrams show only the direct connections that have been defined. Indirect connections that exploit the direct connections can also be defined. A line with a double arrow head is used to represent a peer-to-peer channel and a client-server channel is represented by a line with the arrow pointing to the server. Clients can use the client-server channel both to send messages to the server and to pull messages destined to themselves from that server. Lines with no arrows indicate MQSeries client channels that enable communications between MQSeries Everyplace and MQSeries.

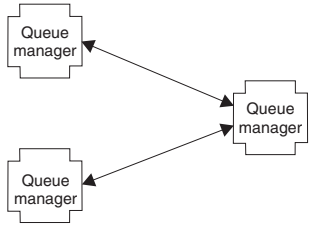
(a) Stand-alone queue manager

(a) Shows a standalone queue manager being used to support one or more applications that use queues to exchange data.



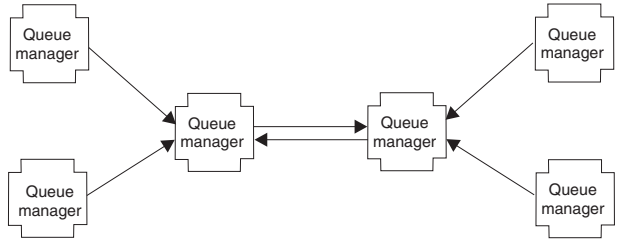
(b) Peer-to-peer configuration

(b) Shows two queue managers interconnected over a peer-to-peer channel.



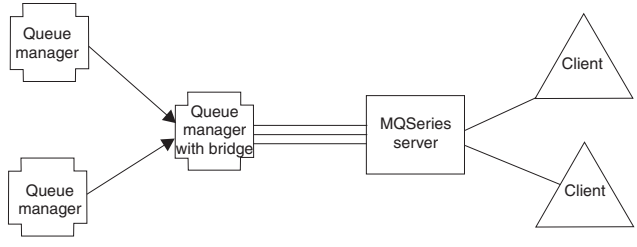
(c) Peer-to-peer configuration

(c) Shows a second direct peer-to-peer channel to a third queue manager. All three can exploit the underlying direct connections to exchange data with each other.



(d) A small network

(d) Shows a small network configuration, where the central server queue managers use a pair of direct client-server channels to exchange information. The client queue managers each use a direct client-server channel to connect with one of the server queue managers.



(e) An integrated MQSeries family network

(e) Shows an MQSeries Everyplace configuration where one of the queue managers has been configured with the bridge option and the pool of client channels have all been directed at a single target MQSeries host/distributed server.

Chapter 4. Product Requirements

This chapter describes the requirements that have shaped the MQSeries Everyplace design and implementation.

Capabilities

MQSeries Everyplace extends the messaging scope of the MQSeries family:

- By supporting low-end devices, such as PDAs, telephones, and sensors, allowing them to participate in an MQSeries messaging network. MQSeries Everyplace also supports intermediate devices such as laptops, workstations, distributed and host platforms. MQSeries Everyplace offers the same quality of service, once only assured delivery of messages, and permits message exchange with other family members.
- By offering lightweight messaging facilities.
- By providing extensive security features to protect messages, queues and related data, whether in storage or in transmission.
- By operating efficiently in hostile communications environments where networks are unstable, or where bandwidth is tightly constrained. MQSeries Everyplace has an efficient wire protocol and automated recovery from communication link failures.
- By supporting the mobile user, allowing network connectivity points to change as devices roam. MQSeries Everyplace also allows control of behavior in conditions where battery resources and networks are failing or constrained.
- By operating through suitably configured firewalls
- By minimizing administration tasks for the user, so that the presence of MQSeries Everyplace on a device can be substantially hidden. This makes MQSeries Everyplace a suitable base on which to build utility-style applications.
- By being easily customized and extended, through the use of application-supplied rules and other classes that modify behavior, or through the sub-classing of the base object classes, for example, to represent different message types.

Applications

Possible MQSeries Everyplace applications are many and varied, but many are expected to be custom applications developed for particular user groups. The following list gives some examples:

- **Retail applications:** trickle feeding of till transactions to host systems, such as message brokers
- **Consumer applications:** supermarket shopping from home using a PDA, gathering of travellers preferences on airlines, financial transactions from a mobile phone
- **Control applications:** collection and integration of data from oil pipeline sensors transmitted via satellite, remote operation of equipment (such as valves) with security to guarantee the validity of the operator
- **Mobile workforce:** visiting professional (insurance agent), rapid publication of proof of customer receipt for parcel delivery companies, fast-food waiter exchanging information with the kitchen, golf tournament scoring, mobile secure systems messaging systems for the police, job information to utility workers in situations where communication is frequently lost, domestic meter reading.
- **Personal productivity:** mail/calendar replication, database replication, laptop downsizing

Customer requirements

Requirements that have influenced the design of MQSeries Everyplace include:

- **Administration:** minimal setup and maintenance; support of both local and remote administration; an ability to extend and customize the administration functions to meet the needs of particular applications; an emphasis on automatic discovery and recovery; the provision of independent administration elements that can be selectively used.
- **Communications:** a very efficient wire protocol; minimal headers; no compulsory fields in messages (excepting a unique identifier); the ability to change the data encoding; compression, encryption and authentication support; end-to-end negotiation of compression and security characteristics; an ability to easily pass through firewalls; pluggable communications adapters.
- **Compatibility:** MQSeries quality of service and seamless messaging interchange; the ability to communicate to existing MQSeries systems without application change; flexible control of message interchange between MQSeries and MQSeries Everyplace .
- **Footprint:** for the Palm device, 64K bytes. For Java devices a minimum classfiles size of 100K bytes.
- **Function:** synchronous and asynchronous messaging capabilities, access to messages held in either local or remote queues; the ability to use any field in the message for selective retrieval; selective control of the backing medium for a queue.

- **Rule support:** control of many aspects of behavior through rules, for example, when to send messages, how often to retry a communication link, what to do with a message that is too big, or how to behave when a target queue is full.
- **Security:** full support for security, authentication and non-repudiation; message-level and queue level security; protection of the messaging system from security attacks; pluggable security using industry standard algorithms; ability to integrate with operating system user credentials; the capability to comply with the national security requirements, allowing security support to change as messages cross country boundaries.

Chapter 5. Product concepts

Introduction

The fundamental elements of the MQSeries Everyplace programming model are messages, queues and queue managers. MQSeries 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 MQSeries Everyplace network. Messages are addressed to a target queue by specifying the target queue manager and queue name pair. Applications place messages on queues through a put operation and typically retrieve them through a get operation. Queues can either be local or remote and are managed by queue managers.Configuration data is stored in a registry.

The MQSeries Everyplace object structure can be seen in the following display from the MQe_Explorer management tool:

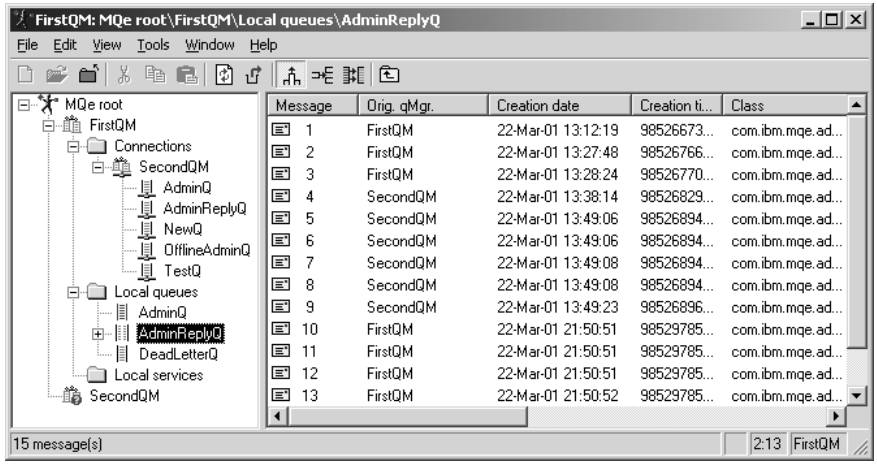


Figure 3. MQSeries Everyplace objects displayed by MQe_Explorer

In the left hand pane the objects are shown in a tree structure. Below the root `MQe root` are two queue managers, `FirstQM` and `SecondQM`. The structure of the `SecondQM` is expanded to reveal folders for *Connections*, *Local queues* and *Local services*. The expansion of *Connections* reveals a connection definition from the `SecondQM` queue manager to the `FirstQM` queue manager. The remote queue manager, `SecondQM` has definitions for five of `FirstQM`'s queues. The expansion of *Local queues* for `SecondQM` reveals three local queues, one of which is selected and has its contents revealed in the right hand pane.

Understanding MQSeries Everyplace requires an appreciation of the nature of messages, queues, queue managers and connections. These concepts are outlined in the following sections.

Message objects

MQSeries Everyplace message objects differ fundamentally from the messages supported by MQSeries messaging. In MQSeries messages are byte arrays, divided into a message header and a message body. The message header is understood and contains vital information, such as the identity of the reply to queue, the reply to queue manager, the message ID, and the correlation ID. The message body is not understood.

In contrast, messages in MQSeries Everyplace are *message objects*, inherited from an MQSeries Everyplace object known as the *fields object*. These messages have no concept of a header or a message body but they do have properties and methods. Understanding the message object first requires an appreciation of the ancestor fields object.

Fields objects, used extensively in MQSeries Everyplace, are an accumulation of *fields*, where a field comprises a name, a data type and the data itself. Field names are Ascii character strings strings (barring a number of reserved characters) of unlimited length.

Fields objects have a *type*, where the type is an abbreviated string corresponding to the programming object class name. Field types may be:

Table 3. Fields object types

Type	Description
Ascii	String or a dynamic array of Ascii strings
Boolean	Value
Byte	Fixed array, or a dynamic array of byte values
Double floating point	Value, fixed array, or a dynamic array of double floating point values
Fields	Object or a dynamic array of fields objects (thus nesting of fields objects is supported)
Floating point	Value, fixed array, or a dynamic array of floating point values
Integer	(4 byte) value, fixed array, or a dynamic array of integers
Long integer	(8 byte) value, fixed array, or a dynamic array of long integers

Table 3. Fields object types (continued)

Type	Description
Object	Value
Short integer	(2 byte) value, fixed array, or a dynamic array of short integers
UNICODE	String or a dynamic array of UNICODE strings

Fields objects support various method calls, for example:

- Enumeration of fields
- Copy a constituent field or fields (with replace option)
- Compare fields objects
- Put and get to and from constituent fields
- Inspect fields objects (for instance, 'is a constituent field contained within the object?')
- Dump and restore to and from a byte array

The dump and restore methods are of particular importance. For example, the dump method is called to provide the data for transmission of a message object over a link, and the restore method allows recreation of that object at the receiver. In this way each object is responsible for its own transmission format, and the default methods provided can be overridden to implement custom behavior. Similarly, these two methods are used when message objects are saved or restored from queues. Overriding these methods can implement very different behavior. For example, a message object may query a database at dump time to extract its value prior to transmission.

Some of the properties of fields objects and their constituent fields are shown in the following table:

Table 4. Fields objects and their constituent field properties

Property	Presence	
	Fields objects	**Fields**
Associated attribute object	Optional	
Constituent field(s)	yes	
Hidden		yes
Name		yes
Type	yes	yes
Value		yes

Two properties of note are the *hidden* property, which allows a field to be ignored for comparison purposes, and the ability to *associate an attributes object* with a fields object.

Attribute objects are fundamental to the MQSeries Everyplace security model and allow selective access to content and the protection of content. They have the following important properties:

Table 5. Attribute object properties

Property	Description
Authentication:	Controls access
Encryption:	Protects the contents when the object is dumped (and allows restoration)
Compression:	Reduces storage requirements (for transmission and/or storage)
Rule:	Controls permitted operations

Message objects inherit from fields objects and thus include all the above capabilities, including the ability to have attributes objects associated at the message and constituent fields object level. Additionally, message objects include a UID (unique identifier) which is generated by MQSeries Everyplace. This UID uniquely identifies each individual message object in the entire MQSeries Everyplace network and is constructed from the:

- **Name** of the originating queue manager (added by the queue manager on receipt of the object). This name must be globally unique.
- **Time** that the message object was created (added at creation)

A message object consequently has the following properties, in addition to those inherited from the ancestor fields object:

Table 6. Message object properties

Property	Description
Originating queue manager	Name of the queue manager that first received the new message from the application
Creation time	Time the message object was created by the application
UID	MQSeries Everyplace unique identifier

No other information is required in a message destined for another MQSeries Everyplace queue manager, though other fields will almost certainly be present. These additional fields arise in various ways:

- Fields added by MQSeries Everyplace to reflect current status

- Fields associated with a particular message subclass
- Custom fields associated with a message object instance

MQSeries Everyplace adds fields to a message object (and subsequently removes them) in order to implement messaging and queuing operations. For example, when a message is on a queue, it is possible to query the message to find out when it was put on that queue. (This is possible using a queue rule which is discussed later). When a message is being sent between queue managers, a resend field may be added to indicate that a retransmission of the data is taking place. There are many other examples.

Typical application-based messages are instances of some descendant of the base message object class and consequently have additional fields as befits their purpose (for example, invoice number). Of these additional fields, some will be generic and common to many applications, such as the name of the reply-to queue manager. In recognition of this MQSeries Everyplace provides support for fields that might be expected to be present in messages. These fields include:

Table 7. Message object fields for which provision is made

Field name	Usage
Action	Used by administration to indicate actions such as enquire, create, delete
Correlation ID	Byte string typically used to correlate a reply with the original message
Errors	Used by administration to return error information
ExpireTime	Time after which message may be deleted (even if it is not delivered)
Lock ID	The key necessary to unlock a message
Message ID	A unique identifier for a message
Originating queue manager	The name of the queue manager that sent the message
Parameters	Used by administration to pass administration details
Priority	Relative priority to be order message transmission
Reason	Used by administration to return error information
Reply-to queue	Name of the queue to which a message reply should be addressed
Reply-to queue manager	Name of the queue manager to which a message reply should be addressed
Resend	Indicates that the message is a resend of a previous message

Table 7. Message object fields for which provision is made (continued)

Field name	Usage
Return code	Used by administration to return the status of an administration operation
Style	Distinguishes commands from request/reply etc.
Wrap message	Wrapped message to ensure data protection

In all cases a defined constant is available that allows the field name to be carried in a single byte. For some fields more extensive provision is made. For example, priority (if present) affects the order in which messages are transmitted, correlation ID triggers indexing of a queue on those field values for fast retrieval, expire time triggers the expiry of the message, and so on.

The administrative panel shown in Figure 4 illustrates the structure of a message. Note that the use of nested fields means that complicated data hierarchies can exists within messages and that the administrative tools (and applications) can drill down into the self-describing structure (provided that this has not been inhibited via security settings).

Message objects use the underlying dump and restore methods when they are serialized to or from a byte array for the purpose of transmission or queue storage. By default, in order to save on footprint, this does not flow the associated class definition. Overwriting dump and restore with standard serialization methods would flow the class definition if this were required.

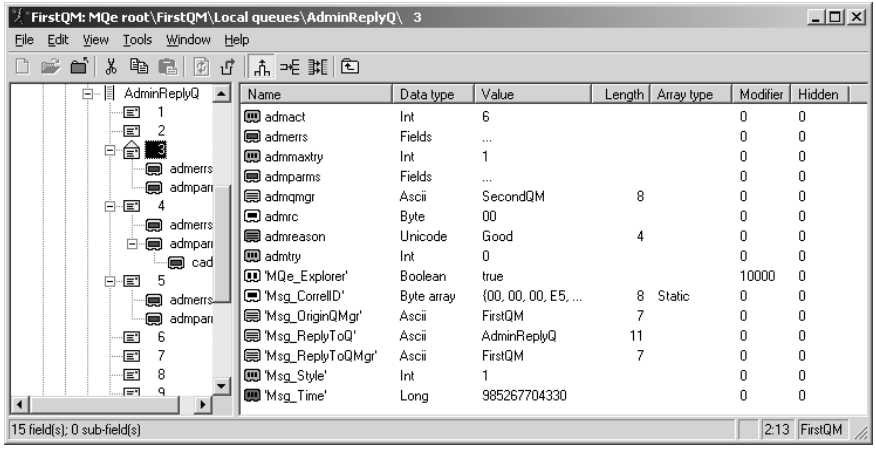


Figure 4. Message structure displayed with MQe_Explorer

Typically however the class is independently made available at each queue manager where the message object is to be instantiated. If it is necessary to

instantiate an object (for example for intermediate storage during transmission) and the class file is known to be unavailable, the message can be wrapped in another class. This is typically the default message object class. This technique is also useful where an attributes object has been used to protect the message. The presence of such an attributes object means that the message contents cannot be accessed without the necessary security keys. However, wrapping such a message allows the kernel message to remain fully protected yet the wrapped message can be freely dumped and restored.

The default message object dump method has been optimized to minimize the size of the generated byte string in order to achieve efficient message storage and transmission.

Dump data format

The default dump data format encodes fields as follows:

```
{Length  Identifier  Fence  {Data}}  {Length  Identifier  Fence  {Data}}  { ...}
```

where:

- *Data:* the data value. Integers are compressed with leading 0s and Fs removed. Booleans have no associated data bytes
- *Fence:* a special byte delimiting the boundary between the identifier and the optional Data item. This byte also indicates the Data item type
- *Identifier:* holds the field name in a variable length Ascii string of bytes, terminated with an end byte
- *Length:* indicates the length of the data field. A variable number of bytes between 1 and 4 are used. The first byte has the first two bits reserved to indicate the length of the length field. Lengths in the range 0 - 1,073,741,823 are supported

This results in a highly compact data stream. Further savings can be achieved by compressing the data. XOR compression with a previous byte stream might be expected to produce good results but, because of the variable nature of these fields and the fact that the order of the fields can change, a simple XOR does not always produce the desired effect. MQSeries Everyplace includes an intelligent XOR, working on a field-by-field basis, that is much more likely to improve compression.

Queues

Queues are typically used to hold message objects pending their removal by application programs. Like messages, queues also derive from the fields objects. Direct access by applications to the queue object is not normally permitted. Instead, the queue manager acts as an intermediary between application programs and queues. Queues are identified by name and the

name can be an ASCII character string of unlimited length[3] but must be unique within a particular queue manager. MQSeries Everyplace supports a number of different queue types:

Local queues

Local queues are used by applications to store messages in a safe and secure manner. The message stored is mapped into permanent storage through an adapter, on a queue-by-queue basis. A range of adapters is supplied with MQSeries Everyplace and others can be written or are available from other sources. The standard adapter is known as the MQeDiskFieldsAdapter that maps queues into the local file system and implements assured delivery. Another adapter, the MQeReducedDiskFieldsAdapter, also maps queues into the file system, but trades faster performance for a dependence on the operating system surviving long enough to empty its buffers into the physical disk subsystem. Yet another adapter, the MQeMemoryFieldsAdapter, maps queues into memory. Whilst giving the fastest queue performance this adapter has the characteristic that messages do not survive a restart of the operating system or queue manager. By creating the appropriate adapter, messages can be stored anywhere, on a queue-by-queue basis. Some examples of storage mediums are a relational database or a writable CD. Adapters exist that take advantage of the mirrored file system on the IBM 4690 retail store controller, or that exploit DB2 for queue storage.

Local queues can be used either on or off-line, (either connected or not to a network). Queues can also have security attributes set, in a very similar manner to the protection of messages and fields objects with the attributes object. Queue security is discussed in “Security” on page 46. Access to messages on local queues is always synchronous, which means that the application waits until MQSeries Everyplace returns after completing the put or get operation.

Remote queues

Remote queues are local references to queues that reside on a remote queue manager. The local reference has the same name as the target queue but the remote queue definition identifies the owning queue manager of the real queue. Remote queues also have properties concerned with access, such as the mode of access (synchronous or asynchronous), any security characteristics and transmission options.

MQSeries Everyplace can establish the remote queues automatically. If an attempt is made to access (for example to send a message to) a queue on another queue manager MQSeries Everyplace looks for a

3. For interoperability it is recommended that the MQSeries naming restrictions are observed, including a maximum name length of 48 characters. The length may also be restricted by the file system you are using.

remote queue definition. If one exists then it is used, but if not *queue discovery* occurs. The characteristics (authentication, cryptography and compression) are discovered and a remote queue definition is created. Such queue discovery depends upon the target being accessible. If the target is not accessible, a remote definition must be supplied in some other way. When queue discovery occurs MQSeries Everyplace sets the access mode to synchronous since the queue is known to be synchronously available.

Synchronous remote queues are queues that can only be accessed when connected to a network that has a communications path to the owning queue manager. If the network is not established then the operations such as put, get, and browse (see "Queue manager operations" on page 36), cause an exception to be raised. The owning queue controls the access permissions and security requirements needed to access the queue. It is the application's responsibility to handle any errors or retries when sending or receiving messages as, in this case, MQSeries Everyplace is no longer responsible for once-only assured delivery.

Asynchronous remote queues are queues that send messages to remote queues but cannot remotely retrieve messages. If the network connection is established then the messages are sent to the owning queue manager and queue. If however the network is not connected, the messages are stored locally until there is a network connection and then the messages are transmitted. This allows applications to operate on the queue when the device is off-line. Consequently these queues have an adapter that maps to a message store in order that messages can be temporarily stored at the sending queue manager whilst awaiting transmission.

Store-and-forward queues

This type of queue stores messages associated with one or more target queue manager destinations. It has two main uses. The first is to enable the intermediate storage of messages in a network, such that they can proceed stepwise to their destination (a forwarding role). The second use is to hold messages awaiting collection (see also home-server queues).

Store-and-forward queues are associated with a set of queue manager names for which they will hold messages. These are referred to as target queue managers. Messages addressed to one of these target queue managers will be placed instead on the relevant store-and-forward queue. The store-and-forward queue may optionally also have a forwarding queue manager name set. If this name is set, the queue attempts to send all its messages to that named queue manager. If the name is not set the queue just holds the messages.

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

Messages on store-and-forward queues are not available to applications.

Home-server queues

Remote queues and store-and-forward queues push messages across the network, the sending queues initiating the transmission. Home-server queues however allow messages to be pulled from a remote queue. A home-server queue definition identifies a store-and-forward queue on a remote queue manager. The home-server queue then pulls any messages that are destined for the home-server queue's local queue manager, off the store-and-forward queue. Multiple home-server queue definitions may be defined on a single queue manager, where each one is associated with a different remote store-and-forward queue.

Home-server queues normally reside on a device and are set up to pull messages from a server whenever the device connects to the network. When the home-server queue pulls a message from the server, the message is then placed in the correct target local queue. Thus the home-server queue does not itself have any application accessible messages. The pull method of getting messages from the server can be more efficient in terms of flows over the network than the server pushing the messages. This is because the home-server queue uses the acknowledgement of the first message as the request for the next message (if any), whereas the server push would require a request/response to send the message and a second request/response for the confirmation flow. A home-server queue normally has a polling interval set that causes it to check for any pending messages on the server whilst the network is connected. This poll interval is an administration configuration option. Home-server queues have an important role in enabling clients to receive messages over client-server channels. The nature of the client-server connection is that servers cannot initiate data transfer. [4]

Messages on home-server queues are not available to applications.

4. The alternative is to use peer-to-peer channels, or to configure both queue managers with both client and server capabilities.

Administration queues
Administration queues are the mechanism through which queue managers (and their associated objects) are configured, either locally or remotely. A message sent to the administration queue is processed by the relevant administration message class and then, optionally, a reply is sent back to the originating application. This topic is discussed in more detail in"Administration" on page 39.

MQSeries bridge queues
This is a specialized form of remote queue with the definition on a gateway and the target queue on an MQSeries queue manager. This form of queue provides a pathway between the MQSeries Everyplace and the MQSeries environments. Transformers are used to perform any necessary data or message reformatting. A basic transformer is supplied with MQSeries Everyplace; programmers are expected to customize this transformer to suit their own requirements.

MQSeries Everyplace stores data securely on queues, ensuring, subject to the adapter, that messages are physically written to the media and not simply buffered by the operating system. However MQSeries Everyplace does not independently log changes to messages and queues. If recovery from media failure is required then hardware solutions must be deployed, such as the use of RAID disk systems. Alternatively the queue must be mapped into recoverable storage such as certain database subsystems.

MQSeries Everyplace does not require that a queue manager has defined queues. However provision is made for four system queues, if required:

- **AdminQ:** required for the receipt of administration messages
- **AdminReplyQ:** optionally used for receiving replies to administration messages
- **DeadLetterQ:** used to store messages that cannot otherwise be delivered
- **SYSTEM.DEFAULT.LOCAL.QUEUE:** a queue that shares a common name with the mandatory system queue on MQSeries servers

Queue properties are shown in the following table. Note however that not all the properties shown apply to all the queue types:

Table 8. Queue properties

Property	Explanation
Admin_Class	Queue class
Admin_Name	Ascii queue name
Queue_Active	Indicates that the queue is active
Queue_AttRule	Rule class controlling security operations

Table 8. Queue properties (continued)

Property	Explanation
Queue_Authenticator	Authenticator class
Queue_BridgeName	Owning MQSeries-bridge name
Queue_ClientConnection	Client connection name
Queue_CloseIdle	Close transporter once all messages have been transmitted
Queue_CreationDate	The date that the queue was created
Queue_Compressor	Compressor class
Queue_Cryptor	Cryptor class
Queue_CurrentSize	Number of messages on the queue
Queue_Description	UNICODE description
Queue_Expiry	Expiry time for messages
Queue_ FileDesc	The location and adapter for the queue
Queue_MaxMsgSize	Maximum length of messages allowed on the queue
Queue_MaxQSize	Max. no. of messages allowed
Queue_Mode	Synchronous or asynchronous
Queue_MQQMgr	MQSeries queue manager proxy
Queue_Priority	Priority to be used for messages (unless overridden by a message value)
Queue_QAliasNameList	Alternative names for the queue
Queue_QMgrName	Queue manager owning the real queue
Queue_QMgrNameList	Queue manager targets
Queue_RemoteQName	Remote MQSeries field name
Queue_Rule	Rule class for queue operations
Queue_QTimerInterval	Delay before processing pending messages
Queue_TargetRegistry	The target registry type
Queue_Transporter	Transporter class
Queue_TransporterXOR	Transporter to use XOR compression
Queue_Transformer	Transformer class

Administrative functions are used to create and delete queues, and to inquire on or modify their properties.

The following MQe_Explorer panels show two of the four property tabs describing the properties of a local queue. Disabled fields indicate that the particular property is not relevant to a queue of that class.

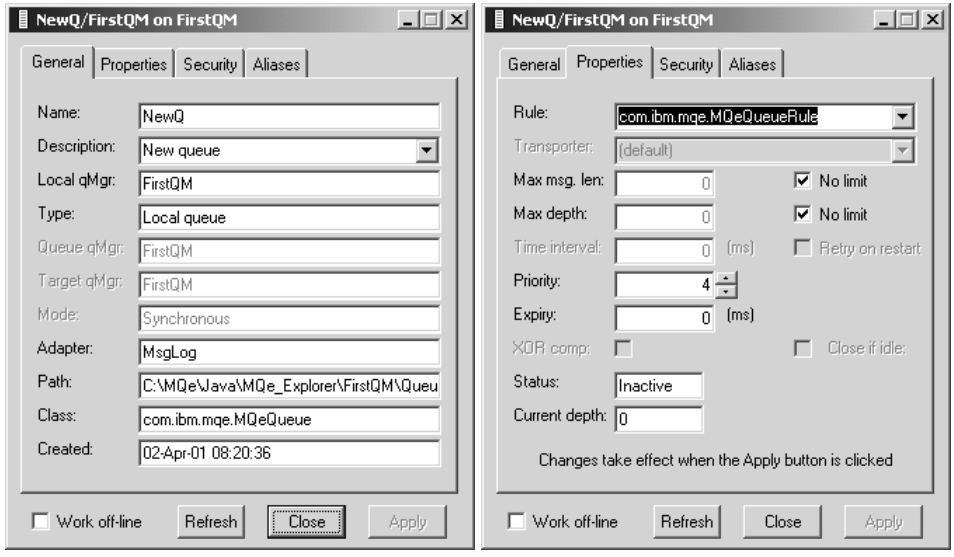


Figure 5. Local queue properties displayed with MQe_Explorer

Queues are not limited to use as a message store. Sub-classed queues can be used in process control application scenarios, for example the queue object could directly control a valve. A message of the right class would cause the valve to be opened, the volume of the flow to be changed etc. An application would not be pulling messages off the queue and performing the action, the queue object would itself controls the action. Other queues could, for example, update spreadsheets or do text to speech conversion. The advantages of this technique are that the security aspects of the queues are still in place and effective, as also is assured messaging. So MQSeries Everyplace would still assure the once-only delivery of the messages, and an associated authenticator and cryptor would guarantee that only the authorized sender of the message could send such messages, with the contents highly secure in transit. No applications would be permitted access to the queue and none would be required.

Queue managers

The MQSeries Everyplace queue manager provides application access to the messages and queues and controls any channels. In MQSeries Everyplace Version 1.2 only one queue manager can be active on a single Java virtual machine at any one time. If there are multiple JVMs on a machine, there can be the same number of queue managers as JVMs. Queue managers are

identified by name and the name must be globally unique[5] and an Ascii character string that can be of unlimited length.[6]

A queue manager that is configured with bridge capabilities, and is therefore able to exchange messages with MQSeries host and distributed products is known as a gateway.

Queue managers can be configured with or without local queueing. All queue managers support synchronous messaging operations; a queue manager with local queueing also supports asynchronous message delivery.

The choice of whether synchronous or asynchronous message delivery is used is determined by the nature of the queue definitions on the sending queue manager. If a synchronous mode remote queue definition exists to the target queue then synchronous delivery is used. If an asynchronous mode remote queue definition exists then asynchronous delivery is used, with the definition providing local storage whilst messages await transmission. If no remote queue definition exists, but a store-and-forward queue exists that handles messages for that target queue manager, then asynchronous delivery is used. In this case, the store-and forward queue provides local storage for messages awaiting transmission. If no queue definitions exist, queue discovery takes place, which if successful, results in synchronous messaging.

Irrespective of whether synchronous or asynchronous messaging is used, MQSeries Everyplace may use either direct or indirect transmission, depending upon the connection definitions available (see “Connections” on page 37). Direct transmission involves just two queue managers, the sending queue manager and the target queue manager. Indirect connection involves a succession of queue managers, with possible protocol changes en route. When indirect transmission is used with synchronous messaging, messaging behavior is unchanged from the direct transmission case. The intermediate queue managers are simply establishing connectivity between the source and the target. However, when indirect transmission is used with asynchronous messaging, the transmission intermediaries may become staging posts for the message as it moves from source to target. Whether they are used in this way depends upon whether suitable intermediate queue storage has been defined on the intermediate queue managers, in the form of appropriate remote queue definitions or store-and-forward queues.

---

5. This restriction is not enforced by MQSeries Everyplace or MQSeries, but duplicate queue manager names may cause messages to be delivered to the wrong queue manager.

6. For interoperability it is recommended that the MQSeries queue manager name rules are observed, including limiting the maximum name length to 48 characters. The length may also be restricted by the file system you are using.

Asynchronous message delivery and synchronous message delivery have very different characteristics and consequences. With asynchronous message delivery the application passes the message to MQSeries Everyplace for delivery to a remote queue. An immediate return is made back to the application. If the message can be delivered immediately (or moved to a suitable staging post) then it is sent, if not it is held locally. Transmission retry logic is defined by the rules that are associated with the queue manager and relevant queues (see "Rules" on page 53). Asynchronous delivery provides once-only assured delivery quality of service, since the message has been passed to MQSeries Everyplace and it has become responsible for delivery.

With synchronous message delivery the application puts the message to MQSeries Everyplace for delivery to the remote queue. MQSeries Everyplace synchronously contacts the target queue and places the message. After delivery MQSeries Everyplace returns to the application. An immediate return is made back to the application. If the message cannot be delivered the sending application receives immediate notification. MQSeries does not assume responsibility for message delivery in the synchronous case.

Synchronous and asynchronous remote queue definitions can be freely established over an MQSeries Everyplace network. When the network also includes MQSeries messaging queue managers (and their associated queues) a number of restrictions are important.

1. Synchronous messaging is not possible to an MQSeries queue manager that is not directly attached to an MQSeries Everyplace gateway (since synchronous messaging is not supported over MQSeries message channels). In order to minimize the consequences of this, the definition of synchronous delivery is changed in this case and it is redefined as being delivery to the MQSeries queue manager directly attached to the gateway. Beyond this queue manager, MQSeries asynchronous messaging is used to enable the message to complete its journey.
2. In MQSeries Everyplace Version 1.2 it is only possible to define a synchronous remote queue definition to an MQSeries messaging remote queue. Taking account the impact of (1) above, this means that asynchronous delivery is not possible (using this definition) to queues on an MQSeries queue manager directly attached to the gateway. However if queuing is required in this case, it can be arranged in the MQSeries Everyplace network by using a second MQSeries Everyplace queue manager, through the use there of appropriate remote queue definitions (or store and forward queues).

Thus asynchronous message delivery means that the local application gives the message to MQSeries Everyplace and its delivery onwards from that local queue manager is the responsibility of MQSeries Everyplace. It means that the network and/or the receiving application need not be available. The time of

the actual delivery is unknown to the sending application. Synchronous message delivery requires the network to be running but the sending application knows that it has been delivered to the receiving application's queue. The receiving application does not need to be available in either the asynchronous or the synchronous case.

MQSeries Everyplace does not offer variations on the once-only assured delivery quality of service for asynchronous messaging as does MQSeries messaging with its persistent and non-persistent options. In MQSeries Everyplace trade-offs between reliability and performance can be made at the queue level through the choice of queue store adapter, for example:

- The MQeDiskFieldsAdapter ensures that data is safely written to disk before processing continues
- The MQeReducedDiskFieldsAdapter ensures that data is passed to the operating system (and can be retrieved) before processing continues
- The MQeMemoryFieldsAdapter saves data in memory

In all cases once-only assured delivery is implemented, however recovery is dependent upon the chosen message store. The performance trade-offs are described in the SupportPac EP01.

Queue managers properties are shown in Table 9.

Table 9. Local queue manager properties

Property	Description
Admin_Class	Queue manager class
Admin_Name	Queue manager name
QMgr_ChnlAttrRules	Channel attribute rules
QMgr_ChnlTimeout	Channel time-out
QMgr_Description	UNICODE description
QMgr_QueueStore	The default location and adapter for queues
QMgr_Rules	Rule class for queue manager operations

The following panel shows one of the two property tabs describing the properties of a queue manager:

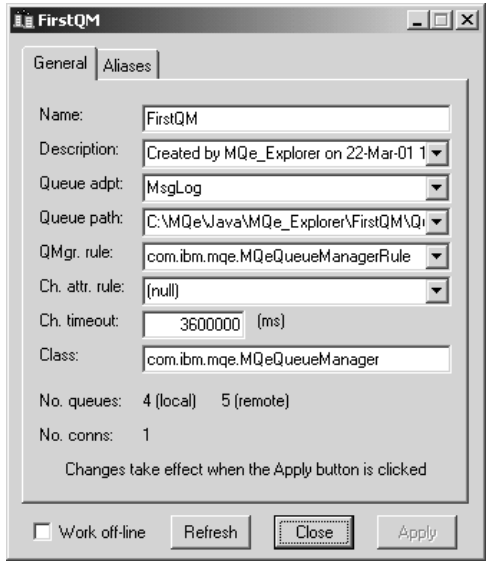


Figure 6. Queue Manager properties displayed by MQe_Explorer

Queue manager configuration

The queue manager runs in an environment that is established by MQSeries Everyplace prior to the queue manager being loaded. The queue manager itself stores its configuration information in its registry (described in more detail in "The registry" on page 50). The queues themselves (containing messages) are stored in queue stores.

The MQSeries Everyplace environment can be established in many ways, either by calls through the API, by utilities shipped with MQSeries Everyplace or through management tools such as the MQe_Explorer. Many of these tools capture the environment parameters in an initialization file, but this is entirely optional. The key environmental parameters are shown in the following table, the section names used follow the conventions used for representing this information in initialization files:

Table 10.

Section name	Property	Explanation
[Alias]	(class alias definitions)	Alias names may be used wherever class names are required
[ChannelManager]	MaxChannels	The maximum number of simultaneous client-server channels to be permitted

Table 10. (continued)

Section name	Property	Explanation
[Listener]	Network	The adapter and port number to be listened on for incoming client/server channel connection requests
	Listen	The adapter to be used to handle resulting connections
	TimeInterval	The client/server channel time-out
[MQBridge]	(initialization parameters for the bridge)	
[MQe_Explorer]	(saved addressing information)	MQe_Explorer saves information which described how incoming connections can be made to this queue manager
[Permission]	(permitted commands)	Permitted channel commands, adapter class and file descriptor maps
[PreLoad]	classes to be loaded when the queue manager is initialized	This provides one mechanism for loading application classes
[QueueManager]	Name	Name of the queue manager
[Registry]	DirName	The registry location
	LocalRegType	The type of registry (file or private)
	PIN	Registry PIN (or prompt request)
	CertReqPIN	Certificate PIN (or prompt request)
	KeyRingPassword	Key ring password (or prompt request)
	CAIPAddrPort	Certificate authority IP address

The following four illustrations show the key tabs in creating a new queue manager using the MQe_Explorer. No prior environment is assumed, and no existing initialization file is required. The result is a running queue manager,

with the configuration data saved in an initialization file so that the queue manager can be restarted simply by opening that file.

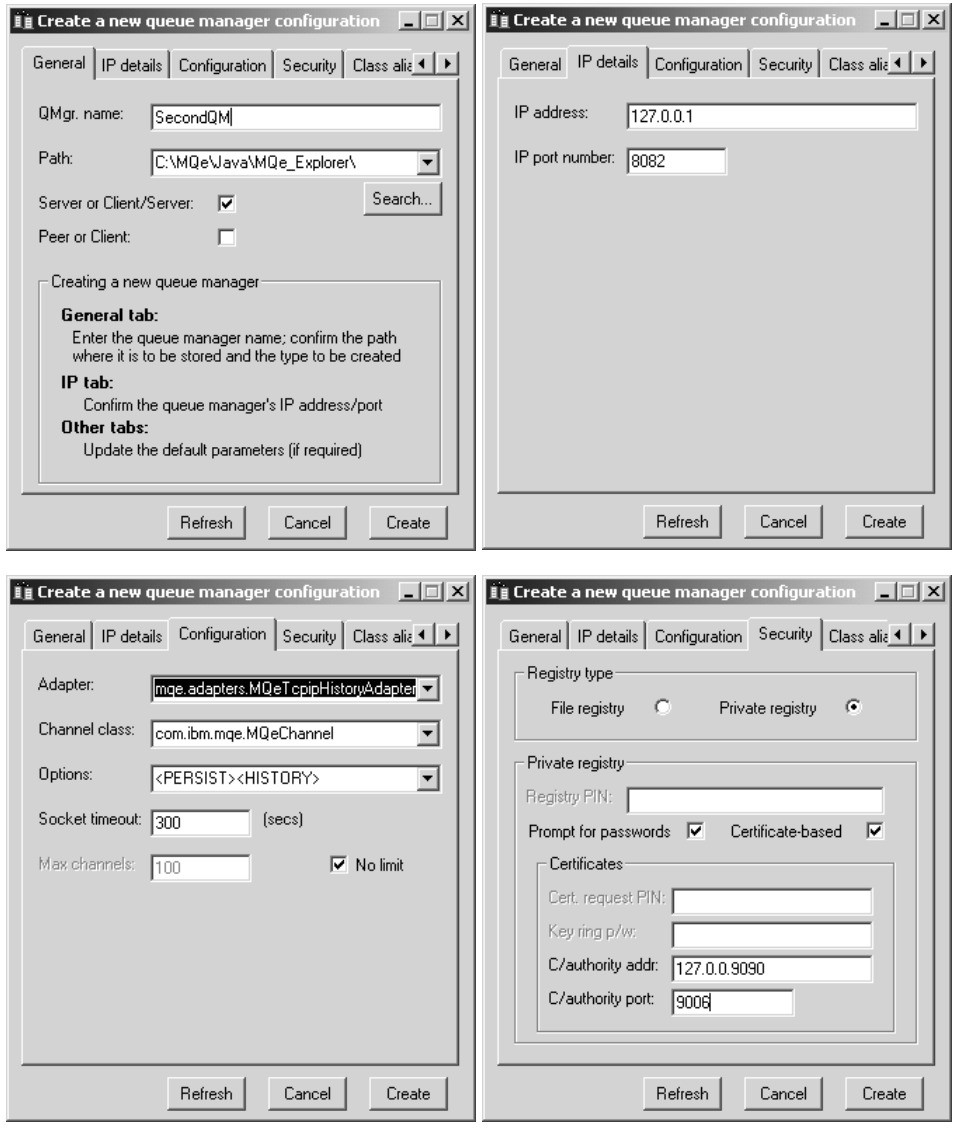


Figure 7. Creating a queue manager with MQe_Explorer

Creating a queue manager does not actually require that the IP address of the queue manager be known. However, MQe_Explorer captures the information for use in configuring other queue managers that to talk to this one. The port number is required in order to listen for incoming client/server connection requests.

The adapter configuration data is needed for the channel listener; the time interval and maximum channels for the channel manager. MQe_Explorer captures the other information as before, to enable other queue managers to be configured.

In this example a secure registry has been configured with certificate-based authentication (see "Security" on page 46). PINs and passwords are disabled because MQe_Explorer prompts for them later, when they are required.

Queue manager operations

Queue managers support messaging operations and optionally manage queues. Applications access messages through the services of the queue manager using methods such as *get*, *put*, *browse*, *wait*, *listen* and *delete*. Many of these operations take a *filter* as one of their parameters. A filter is a fields object that is matched for equality and any fields in the message can be used for selective retrieval. Most method calls also include an attribute object to be used in the encoding or decoding of a message.

The get operation destructively removes messages from a queue. Subject to the conditions imposed by the filter, messages are retrieved in priority order and, within that, in the order of their time of arrival on the queue. So, all other things being equal, the first message to arrive will be the first to be retrieved. Get is available as a single step or two step operation. The two-step case is provided for use in those situations where it is essential that there is no possibility of message loss as the messages passes from MQSeries Everyplace to the application. First a get is issued with a *confirm ID* (its value being chosen by the application). That operation gets the message for the application but instead of deleting it from the queue immediately it hides it on the queue. A subsequent confirm operation, specifying the original message UID, indicates that the get was successful for the application, and it is then that the message is deleted. Failure of the get allows the message to be recovered. Put operations behave in a similar way.

By specifying the UID, messages can be *deleted* from a queue without being retrieved.

If nondestructive read is required, queues may be *browsed* for messages (optionally under the control of a filter). Browsing retrieves all the message objects that match the filter, but leaves them on the queue. *Browsing under lock* is also supported. This has the additional feature of locking the matching messages on the queue. Messages may be locked individually, or in groups identified through a filter, and the locking operation returns a *lock ID*. Locked messages can be got or deleted only if the lock ID is supplied. An option on browse allows either the full messages, or only the UIDs, to be returned.

Applications can *wait* for a specified time for messages to arrive on a queue. Optionally a filter can be used to identify those of interest and a *confirm ID* can also be specified. Alternatively applications can listen for MQSeries Everyplace message events, again optionally with a filter. Listeners are notified when messages arrive on a queue.

Queues are enabled for messaging operations as shown in Table 11

Table 11. Messaging operations on MQSeries Everyplace queues

	Local queue	**Remote queue[1]**	
		Synchronous	**Asynchronous**
Browse (±lock, ±filter)	Yes	Yes	
Delete	Yes	Yes	
Get (±filter)	Yes	Yes	
Listen (±filter)	Yes		
Put	Yes	Yes	Yes
Wait (±filter)	Yes	Yes	

Notes:

1. The synchronous remote wait operation is implemented through a poll of the remote queue, so the actual wait time is a multiple of the poll time
2. [1]The MQSeries Everyplace MQSeries Bridge supplied with MQSeries Everyplace Version 1.2 only supports assured/unassured put, unassured get, and unassured browse(without lock).

Connections

Topology and access through the MQSeries Everyplace network is defined by connection objects. These definitions are stored locally at each queue manager. They are created, modified and destroyed through the standard administrative provision.

A connection object typically defines the access to a remote queue manager (and consequently they are sometimes referred to as remote queue manager definitions). The properties are given in the following table:

Table 12. Connection (remote queue manager) properties

Property	**Explanation**
Admin_Name	Queue manager name
Con_Adapter	The adapter file descriptor
Con_AdapterOptions	Adapter options (such as use history)

Table 12. Connection (remote queue manager) properties (continued)

Property	Explanation
Con_AdapterParm	ASCII data to be use by an adapter (such as servlet name)
Con_Aliases	Alternative names for the queue manager/connections
Con_Channel	The type of channel that this connection should use
Con_Description	UNICODE description
Queue_QMgrName	Owner of the definition

The following administration panels show two of the tabs associated with a connection definition:

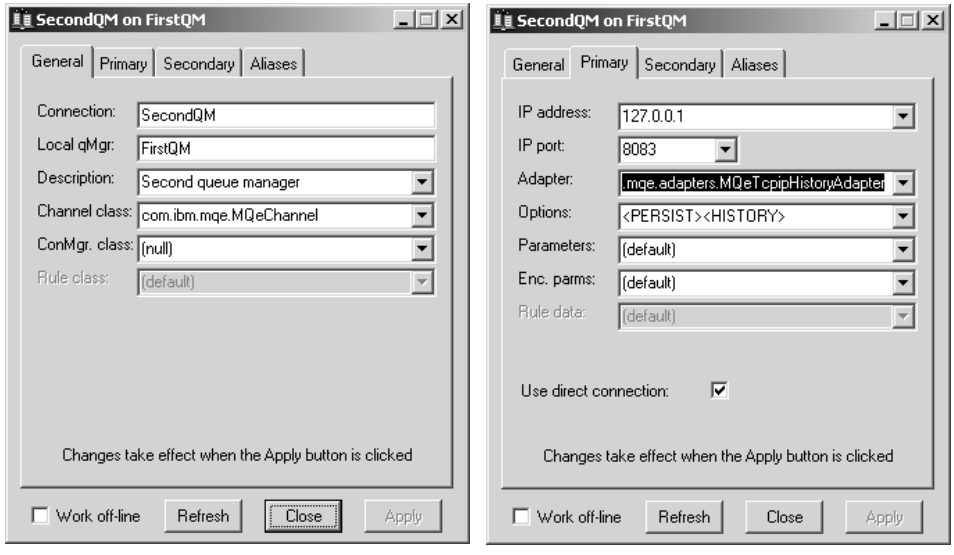


Figure 8. Connection definition displayed with MQe_Explorer

Data can be passed to the chosen communications adapter in the form of options, parameters and encoded parameters.

Indirect connections can be specified. In this case MQSeries Everyplace routes the connection through other queue managers (which can be chained), and the protocol may change en route. Indirect connections are particularly useful in enabling devices to have a single point of entry to an MQSeries Everyplace network.

Connection objects are also used to define listeners for incoming peer-to-peer channels.

As for most MQSeries objects, aliases can be defined for connections. A local connection (defined as a connection with a name matching that of the local queue manager) is used to define alias names for the local queue manager itself.

Administration

Administration provides facilities to configure and manage MQSeries Everyplace resources such as queues and connections. Message-related functions are regarded as the responsibility of applications. Administration is enabled through an interface that handles the generation and receipt of administrative messages and is designed so that local and remote administration is handled in an identical manner. Requests are sent to the administration queue of the target queue manager and replies may be received if required. Any local or remote MQSeries Everyplace application program can create and process administration messages directly or indirectly through helper methods. Administration messages can also be generated indirectly through the MQe_Explorer[7], a management tool that provides a graphical user interface for system administration.

The administration queue does not understand how to perform the administration of individual resources; this knowledge is encapsulated in each resource and its corresponding administration message.

Administration messages

Administration messages extend the base MQSeries Everyplace message object.Table 13 lists the message classes provided for administration of MQSeries Everyplace resources. These base administration messages can be sub-classed to provide for the administration for other objects; for example a different type of queue could be managed using a subclass of MQeQueueAdminMsg. The MQSeries Everyplace bridge to MQSeries uses subclasses of the MQeAdminMsg in this way.

Table 13. Administration message classes

Administration message class	Use
MQeAdminMsg	Abstract class used as the basis of all administration messages
MQeQueueManagerAdminMsg	Administration of queue managers
MQeQueueAdminMsg	Administration of local queues

7. MQe_Explorer is not included with MQe_Explorer but is available for free download from the MQSeries Everyplace site on the World Wide Web (*http://www.ibm.com/software/ts/mqseries/everyplace*).

Table 13. Administration message classes (continued)

Administration message class	Use
MQeRemoteQueueAdminMsg	Administration of remote queues
MQeAdminQueueAdminMsg	Administration of the administration queue
MQeHomeServerQueueAdminMsg	Administration of home server queues
MQeStoreAndForwardQueueAdminMsg	Administration of store and forward queues
MQeConnectionAdminMsg	Administration of connections between queue managers
MQeClientConnectionAdminMsg	Administration of a bridge client connection object, used to connect to MQSeries
MQeListenerAdminMSg	Administration of a bridge transmission queue listener object, used to collect messages from MQSeries
MQeBridgeAdminMsg	Administration of a bridge to MQSeries
MQeMQBridgesAdminMsg	Administration of a list of MQSeries-bridges
MQeMQQMgrProxyAdminMsg	Administration of a bridge representation of an MQSeries queue manager
MQeMQBridgeQueueAdminMsg	Administration of an MQSeries-bridge queue

The structure of an administration message depends upon its particular class, that is the nature of the resource that it is managing, and the details of the operation to be performed on that resource. Generically however, the administration messages are structured as shown in Table 14:

Table 14. Generic structure of an administration message

Level 1 fields	Level 2 and below fields	Use
Admin_Action		Create, delete, inquire, etc.
Admin_Errors		Fields object parent
	Multiple fields	Detailed information on a per-error basis
Admin_MaxAttempts		Maximum number of times the administration action should be attempted

Table 14. Generic structure of an administration message (continued)

Level 1 fields	Level 2 and below fields	Use
Admin_Parameters		Fields object parent
	Resource	Name of resource to be managed
	Multiple fields	Detailed parameter data specific to the message class and action
Admin_Reason		Text message indicating reason for failure
Msg_ReplyToQ		Name of the queue to which the response should be sent
Msg_ReplyToQMgr		Name of the queue manager to which the response should be sent
Admin_RC		Numeric return code indicating the outcome
Msg_Style		Command or request/reply
Admin_TargetQMgr		Name of the queue manager owning the target resource

Three styles of administration message are supported, namely commands (datagrams) that indicate an administration action that does not require a reply, requests that require a reply, and the replies themselves. The reply is constructed from a copy of the original message; thus additional fields can be added by the sender for use by the receiver.

Selective administration

Access to administration can be controlled through the authenticator on the administration queue. For local applications the supplied authenticator considers them all to represent the same local user and therefore either allows or disallows administration for them all. Remote administration applications are controlled by the invocation of the authenticator on the channel before any administration messages flow. Different remote users can thus be distinguished and separately enabled or disabled. In all cases for any user, administration is enabled or disabled in its entirety. If a finer level of administration control is required, for example certain administration users are to be given access to some queues and not others, then additional programming is required. A more sophisticated authenticator can keep track of permissions associated with user identities, and administration messages can be subsequently be processed on the basis of these permissions (see

"Security" on page 46). Rules associated with queues can also be exploited to allow or disallow actions in a similar manner (see "Rules" on page 53).

Monitoring and related actions

Administration often embraces more than object creation and modification. It can include monitoring a system and informing an operator when a queue is full, or dealing with an error situation such as by taking appropriate action when a message arrives that is too large for its target queue. These aspects are handled in MQSeries Everyplace through the use of rules, that is classes that are invoked whenever objects significantly change their status or when certain types of error situations arise. A default set of rules classes is provided with MQSeries Everyplace but typically these are replaced with custom classes (see "Rules" on page 53).

Dynamic channels

MQSeries Everyplace communicates between queue managers through logical links known as dynamic channels. These support bidirectional flows and are established by the queue manager as required. Asynchronous and synchronous messaging both use the same channels and the protocol used is unique to MQSeries Everyplace. This protocol can be customized on a per message basis by overriding the message dump method. By contrast MQSeries usually uses client channels for its synchronous traffic and a pair of message channels for bidirectional asynchronous messaging. MQSeries *cluster message channels* have some similar characteristics to the MQSeries Everyplace dynamic channels, but there are many important differences.

A dynamic channel is a logical connection between two queue managers, established for the purpose of sending or receiving data. Multiple concurrent channels can exist, even between the same parties. They have characteristics, for example authentication, cryptography, compression, and the transport protocol used. These characteristics are pluggable, (different versions may be used on different channels) and consequently each channel has its own quality of service attributes of:

- **Authenticator:** either null or an *authenticator* object that can perform user or channel authentication
- **Channel:** the class providing the transport services.
- **Compressor:** either null or a *compressor* object that can perform data compression and decompression
- **Cryptor:** either null or a *cryptor* object that can perform encryption and decryption
- **Destination:** the target for this channel, for example SERVER.XYZ.COM

The authenticator is typically only used when setting up the channel. Compressors and cryptors are typically used on all flows.

The simplest type of cryptor is MQeXorCryptor, which encrypts the data being sent by performing an exclusive-OR of the data. This encryption is not secure, but it modifies the data so that it cannot be viewed. In contrast, MQe3DESCryptor implements triple DES. The simplest type of compressor is the MQeRleCompressor, which compresses the data by replacing repeated characters with a count. Other authenticators, compressors, and cryptors are supplied, see Table 15 on page 47.

Channel establishment uses protocol adapter specifications to determine the links and protocols to be used for a particular channel. At each intermediate node the channel definitions are searched to resolve the addressing needed for the next link. Where no onwards definition exists, the channel ends and any messages flowing through are passed to the queue manager at that point. In this way, intermediate store-and-forward queues and remote queue definitions can be exploited.

Channels are not directly visible to applications or administrators and are established by the queue manager as required. Channels link queue managers together and their characteristics are negotiated and renegotiated by MQSeries Everyplace dependent upon the information to be flowed. Transporters are the MQSeries Everyplace components that exploit channels to provide queue level communication. Again, these are not visible to the application programmer or administrator.

When assured messaging is demanded MQSeries Everyplace delivers messages to the application once, and once-only. It achieves this by ensuring that a message has successfully passed from one queue manager to another, and been acknowledged, before deleting the copy at the transmitting end. In the event of a communications failure, if an acknowledgment has not been received, a message may be retransmitted (once-only delivery does not imply once-only transmission) but duplicates are not delivered.

Adapters

Adapters are used to map MQSeries Everyplace to device interfaces. Channels exploit the protocol adapters to run over HTTP, native TCP/IP, UDP, and other protocols. Similarly queues exploit fields storage adapters to interface to a storage subsystem such as memory or the file system Adapters provide a mechanism for MQSeries Everyplace to both extend its device support and to allow versioning.

A *file descriptor* is a string that is used to identify, load and activate an adapter.

Dialup connection management

Dialup networking support for devices is handled by the device operating system. When MQSeries Everyplace on a disconnected device attempts to use the network, for example because a message must be sent, then if the network stack is not active, the operating system itself initiates remote access services (RAS). Typically this takes the form of a panel displayed to the user, offering a dialup connection profile. Until the connection is established, the operating system is in control. Consequently the device user must ensure that appropriate dialup connection profiles are available for the operating system to use. There is no explicit support for dialup networking in MQSeries Everyplace Version 1.2.

Trace

Trace is enabled by running an independent program that performs tracing actions. Embedded within MQSeries Everyplace are calls to trace for information, warning and error situations with system and user variants. Applications may also call trace directly and may add new messages or modify existing trace messages. The supplied sample trace program allows selected messages to be displayed, printed and/or directed to the event log. Other trace programs can be written with additional capabilities or be designed to format and deliver their output in other ways.

Most MQSeries Everyplace exceptions are passed to the application for handling, and the application exception handler may also route these to trace.

Event log

MQSeries Everyplace provides event log mechanisms and interfaces that may be used to log status, queue manager started for example. Logging can be initiated and by default written out to a file, however this can be intercepted and directed elsewhere. The MQSeries Everyplace event log does not log message data and cannot be used to recover messages or queues.

Message delivery

MQSeries Everyplace networks are connected queue managers and may include gateways. They can span multiple physical networks and route messages between them. In general they provide synchronous and asynchronous access to queues with a programming model that is independent of queue location.

Asynchronous message delivery

When a message is asynchronously put to a remote queue, the message object is logically placed on the backing store associated with the local definition of that queue, along with its destination queue manager and queue names, and

with the compressor, authenticator and cryptor characteristics that match the target destination of the message. The object's dump method is called as the object is saved to persistent storage in a secure format, as defined by its destination queue. The queue manager controls message delivery. It identifies (or establishes) a channel with appropriate characteristics to the queue manager for the next hop, then creates (or reuses) a transporter to the target queue. The transporter dumps the object and transmits the resulting byte string. Note that the target queue manager and queue name are not part of that message flow.

If appropriate, the message is encrypted and compressed over the channel. If it has reached its destination queue manager, it is decrypted and decompressed. A new message object is created, using the restore method of that object class, with the resultant object being placed on the destination queue. If the message has not reached its destination queue manager, it is decrypted and decompressed, then placed on a store and forward queue with the appropriate characteristics for onwards transmission. In both cases it is held on its respective queue in a secure format, as defined by its destination queue.

A characteristic of asynchronous message delivery is that messages are passed to the queue manager at intermediate hops, being queued for onwards transmission. Messages are taken off the intermediate queues firstly in priority order, then in timestamp sequence.

Synchronous message delivery

Synchronous message delivery is similar to the asynchronous case described above, but the queue manager involvement in intermediate hops takes place at a much lower level, involving the transporter and channels. A channel is established end-to-end, using the adapters defined in the protocol specifications at each intermediate node, to identify the next link. At the end of the last link, where no further relevant file descriptors exist, the message gets passed to the higher layers of the queue manager for processing. Thus the sending node does not queue the message but passes it along the channel, through intermediate hops, and then gives it to the destination queue manager to place it on the target queue.

The link into MQSeries uses a bridge queue on the gateway, which transforms the message to an MQSeries format. This mechanism means that synchronous MQSeries Everyplace -style messaging from a device is possible to MQSeries, with the dynamic channel terminating at the gateway. The message is delivered in real time from the gateway, through a client channel, to an MQSeries server. From there its destination may require it to be routed asynchronously along MQSeries message channels

In a similar manner a device capable of only synchronous messaging can send messages to an asynchronous MQSeries Everyplace queue, provided that a suitable intermediary is available.

Security

MQSeries Everyplace provides an integrated set of security features enabling the protection of message data both when held locally and when it is being transferred.

MQSeries Everyplace security features provide protection in three different categories:

- Local security - local protection of message (and other) data
- Queue-based security - protection of messages between initiating queue manager and target queue
- Message-level security - message level protection of messages between initiator and recipient

MQSeries Everyplace local and message-level security are used internally by MQSeries Everyplace, but are also made available to MQSeries Everyplace applications. MQSeries Everyplace queue-based security is an internal service.

The MQSeries Everyplace security features of all three categories protect message data by use of an attribute (MQeAttribute or descendent). Depending on the category, the attribute is applied either explicitly or implicitly.

Each attribute can contain the following objects:

- Authenticator
- Cryptor
- Compressor
- Key
- Target Entity Name

These objects are used differently, depending on the category of MQSeries Everyplace security feature, but in all cases, the MQSeries Everyplace security feature's protection is applied when the attribute attached to a message object is invoked. This occurs when an MQSeries Everyplace message's dump method is invoked (when the attribute's encodeData method is used, for example to encrypt and compress the message data). The MQSeries Everyplace security feature's unprotect occurs when the MQSeries Everyplace message's restore method is invoked (when the attribute's decodeData method is used, for example to decompress and decrypt the message data).

The algorithms supported by MQSeries Everyplace Version 1.2 for authentication, encryption and compression are detailed in Table 15.

Table 15. Authentication, encryption and compression support

Function	Algorithm
Authentication	WTLS mini-certificate
	Validation Windows NT/2000, AIX, or Solaris identity
Compression	LZW
	RLE
Encryption	Triple DES
	DES
	MARS
	RC4
	RC6
	XOR

MQSeries Everyplace local security

Local security protects MQSeries Everyplace message (or MQeFields or MQeFields descendent) data locally. This is achieved by creating an attribute with an appropriate symmetric cryptor and compressor, creating and setting up an appropriate *key* (by providing a password or passphrase). The key is explicitly attached to the attribute, and the attribute is attached to the MQSeries Everyplace message. MQSeries Everyplace provides the MQeLocalSecure class to assist with the setup of local security, but in all cases it is the responsibility of the local security user (MQSeries Everyplace internally or an MQSeries Everyplace application) to set up an appropriate attribute and manage the password or passphrase key.

MQSeries Everyplace queue-based security

Queue-based security can be applied to synchronous and asynchronous messages.

Synchronous queue-based security

Use of synchronous queue-based security allows an application to leave all message security considerations to MQSeries Everyplace . Queues have authentication, encryption and compression characteristics and these are used to determine the level of security needed to protect message flows (as well as for persistent storage).

When a message is to be sent, the security characteristics of the target queue are retrieved from the local registry. If these are not present , the queue manager attempts to discover the target characteristics from the target queue

manager and caches them for subsequent reuse. If a channel exists to that queue manager it is used; if not, a new channel is created. The target queue attributes are retrieved.

Based on the quality of service required, the channel attributes to the target queue manager are dynamically changed. This is subject to any rules that have been established. Typically a rule allows an upgrade in the level of security, (for example from no protection to weak protection, or from weak to strong). If the channel cannot be upgraded, or the security level is deemed excessive (for example no protection is required and the available channel implements strong protection) then a new channel is created. A pool of channels exists, reused where possible, with dynamically changing characteristics according to the demands of the traffic. Channels are automatically destroyed when not required. Messages are always placed on queues at the security level defined by the target queue characteristics.

Authentication takes place at the channel level, keeping the overhead per message to a minimum. Synchronous queue-based security is also typically used with symmetric cryptors since this results in fast encryption/decryption. However, in these symmetric cases, MQSeries Everyplace uses RSA asymmetric encryption initially, to protect the flows necessary to establish a shared key at the sender and receiver. After that point symmetric encryption is used to protect the confidentiality of the data flowed. MQSeries Everyplace makes the cryptographic attack of this data more difficult by changing the key dynamically on each channel flow. MQSeries Everyplace also ensures the integrity of the data flowed by generating and appending the digest to the data before sending, and regenerating and validating it on receipt.

Asynchronous queue-based security

Asynchronous messaging differs from the synchronous case described above in as far as there is no guarantee that the target queue is accessible at the time the putMessage is executed. In this case the queue manager cannot send the message immediately and places it on the transmission queue; however it is encrypted in accordance with its target queue characteristics. When it can be transmitted, it is decrypted, and then sent down a channel with suitable characteristics. Thus messages are always protected, even while awaiting transmission. Asynchronous messaging requires a remote queue definition to enable the target queue characteristics to be determined.

In the asynchronous case, authentication is not possible between originator and target. Where authentication is important, for example for a recipient to determine the message's originator (to determine acceptance or establish non-repudiation) or for an initiator to ensure that message can only be processed by the intended recipient, message-level security must be used.

Queue-based security can be used at the same time as message-level security, but it is not necessary, since message data is already protected.

Message-level Security

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

Message-level security is an application layer service that requires the initiating MQSeries Everyplace application to set up a message-level attribute and provide it when using putMessage to put the message to a target queue. The receiving application must set up and pass a matching message-level attribute to the receiving queue manager so that the attribute is available when the application invokes getMessage to get the message from the target queue.

Like local security, message-level security exploits the application of an attribute on a message object. The initiating application's queue manager handles the putMessage with the dump method, which uses the attribute's encodeData method to protect the message data. The receiving application's queue manager handles the application's getMessage with the restore method which uses the attribute's decodeData method to recover the original message data.

MQSeries Everyplace supplies two alternative attributes for Message-level security:

MQeMAttribute
: This is used for business-to-business communications where mutual trust is tightly managed in the application layer and requires no trusted third party. All available MQSeries Everyplace symmetric cryptor and compressor choices can be used. Like local security, the attribute's key must be preset before it is provided with putMessage or getMesssage. MQeAttribute provides a simple and powerful method for message-level protection enabling the use of strong encryption to protect message confidentiality, without the overhead of any public key infrastructure (PKI).

MQeMTrustAttribute
: This attribute provides a more advanced solution using digital signatures and exploiting the default public key infrastructure. It uses ISO9796 digital signature/validation to enable the receiving application to establish proof that the message comes from the purported sender. The supplied attribute's cryptor is used to protect 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, all available MQSeries Everyplace symmetric cryptor and compressor

choices can be used. Chosen for size optimization, the certificates used are WTLS mini-certificates. The mutual availability of the information necessary to authenticate (validate signatures) and encrypt/decrypt is provided through the MQSeries Everyplace default infrastructure.

A typical MQeMTrustAttribute 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
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

Message-level security is independent of queue-level security.

The registry

The registry is the primary store for queue manager-related information and 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)

Access to the registry is normally restricted to the legitimate queue manager user and is PIN protected, but a configurable option enables this to be bypassed by users more concerned with footprint size than security.

MQSeries Everyplace Authenticatable entities

Queue-based security, which uses mini-certificate based mutual authentication, and message-level protection, which 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 (people), but in general, messaging has no concept of a user. Usually this concept is managed at the application level, that is, by the user of messaging services. MQSeries Everyplace deliberately abstracts the concept of *target of authentication* from *user* to *authenticatable entity*. This does not exclude the possibility of authenticatable entities being people, but this would be an application

selected mapping. Internally, MQSeries Everyplace defines all queue managers that can either originate or be the target of mini-certificate dependent services as authenticatable entities. In addition, MQSeries Everyplace also defines queues that are defined to use mini-certificate based authenticators to be authenticatable entities. So a queue manager that supports these services may have one authenticatable entity, the queue manager, or a set of authenticatable entities, the queue manager and every queue that uses certificate based authenticator.

Private Registry and 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. Classically, these challenges are more difficult to solve than the underlying cryptographic techniques. MQSeries Everyplace provides default services that can be used to enable authenticatable entities to perform auto-registration. Private registry (a descendent of base registry) to enable secure management of an authenticatable entity's private credentials, and public registry (also a descendent of base registry) to manage set of public credentials. The private registry provides a base registry with many of the qualities of a secure or cryptographic token, for example, it can be a secure repository for public objects like mini-certificates, and private objects like private keys. It provides a mechanism to allow only the authorized user to access the private objects. It provides support for services (for example digital signature, RSA decryption) in such a way that the private objects never leave the private registry. By providing a common interface, it hides the underlying device support, which is currently is restricted to the local file system, but may well be extended to map to portable tokens in the future.

Auto-registration

MQSeries Everyplace provides default services that support auto-registration. These services are automatically triggered when an authenticatable entity is configured, for example when a queue manager is started or when a new queue is defined. In both cases registration is triggered and 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, it 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 its new credentials. While auto-registration provides a simple mechanism to establish an authenticatable entity's credentials, for message-level protection (MqeMTrustAttribute, see above), access to the intended recipient's public key (mini-certificate) is also required.

Public registry and certificate replication

MQSeries Everyplace provides default services that enable the sharing of authenticatable entity public credentials (mini-certificates) between MQSeries Everyplace components. These are a prerequisite for MQeMTrust based message-level security. MQSeries Everyplace public 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, instead of phone numbers, it is a set of mini-certificates of the authenticatable entities that are the most frequently contacted. The public registry is not purely passive in its services. If accessed to provide a mini-certificate that it does not hold, and if configured with a valid home-server component, the public registry automatically attempts to fetch the requested mini-certificate from the public registry of the home-server. These services can be used to provide an intelligent automated mini-certificate replication service, that facilitates the availability of the right mini-certificate at the right time.

Application use of registry services

While the MQSeries Everyplace queue manager is designed to exploit the advantages of using private and public registry services, access to these services is not restricted. MQSeries Everyplace solutions may wish to define and manage their own authenticatable entities, for example users. Private-registry services can then be used to auto-register and manage the credentials of the new authenticatable entities, and public-registry services are used to make the public credentials available where needed. All registered authenticatable entities can be used as the initiator or recipient of message-level services protected using MQeMTrustAttribute

Default mini-certificate issuance service

MQSeries Everyplace provides a default mini-certificate issuance service that can be configured to satisfy private-registry auto-registration requests. With the tools provided with MQSeries Everyplace , a solution can setup and manage a mini-certificate issuance service to issue mini certificates to a carefully controlled set of entity names. The characteristics of this issuance service are:

- Management of the set of registered authenticatable entities
- mini-certificate issuance
- WAP WTLS mini-certificate Repository management

The tools provided with MQSeries Everyplace enable a mini-certificate issuance service administrator to authorize mini-certificate issuance to a given entity by registering its entity name and registered address and defining a one-time-use certificate request PIN. This is normally done after offline checking has validated the authenticity of the requestor. The certificate request PIN is posted to the intended user (for example in a similar way to the way

that bank card PINs are posted to users when a new bank card is issued). The user of the private registry (for example the MQSeries Everyplace Application or MQSeries Everyplace queue manager) can then be configured to provide this certificate-request PIN at startup time. When the private registry triggers auto-registration, the mini-certificate issuance service validates the resulting new-certificate request (based on a match of the presented entity name and certificate-request PIN with their preregistered values), issues the new mini-certificate and resets the registered certificate-request PIN so that it cannot be reused. All auto-registration new mini-certificate requests are processed on a secure channel.

The set of mini-certificates issued by a mini-certificate issuance service is held in the issuance service's own registry. When a mini-certificate is reissued (for example as the result of expiry) then the expired mini-certificate is archived.

The security interface

An optional interface is provided that may be implemented by a custom security manager. The methods allow the security manager to authorize or reject requests associated with:

- Addition or removal of class aliases
- Definition of adapters
- Mapping of file descriptors
- Processing of channel commands

Customization

Rules

Rules are Java classes that are used to customize the behavior of MQSeries Everyplace when various state changes occur. Default rules are provided where necessary, but these may be replaced with application- or installation-specific rules to meet customer requirements. The rule types supported differ in how they are triggered, not what they can do. Rules contain logic and can therefore perform a wide range of functions.

Attribute rules

This rule class is given control whenever change of state is attempted, for example, a change of:

- Authenticator
- Compressor
- Cryptor

The rule would normally allow or disallow the change.

MQSeries bridge rules

These rules classes are given control when the MQSeries Everyplace to MQSeries-bridge code has a change of state. There is a separate bridge rule class to determine each of the following:

- What to do with a message when a listener cannot deliver it onto MQSeries Everyplace, when it is coming from MQSeries. For instance because the message is too big, or the queue does not exist.
- The state that bridge administered objects should start in once the server is instantiated
- What to do when the bridge finds something wrong with the Sync queue on MQSeries (the persistent store used for crash recovery). The default rule just displays the problem.
- How to convert an MQSeries Everyplace message to an MQSeries message, and vice-versa. Transformers to do message conversion between MQSeries Everyplace and MQSeries messages are not derived from any MQeRule classes, instead they must implement the MQeTransformerInterface interface. Apart from this, transformers act like rules and are invoked when a message requires format conversion.

Queue rules

This rule class is given control whenever a change of state of the associated queue occurs, for example:

- Adding a message to a queue. For example to see if a threshold is exceeded (number of messages, size of message, invalid priority)
- Queue characteristics assigned or changed
- Queue is opened or closed
- Queue is to be deleted

Queue manager rules

This rule class is given control whenever a change of state of the queue manager occurs, for example:

- Queue manager is opened. For example, start a background timer thread running to allow timed actions to occur
- Queue manager is closed. For example terminate the background timer thread
- A new queue is added

Connection styles

MQSeries Everyplace can support client-server and peer-to-peer operation. A *client* is able to initiate communication with a server. A *server* is only able to respond to the requests initiated by a client. In *peer-to-peer* operation, the two peers can initiate flows in either direction. These connection styles require different components of MQSeries Everyplace to be available and active. The components involved are:

- **Channel listener:** that listens for incoming connection requests.
- **Channel manager:** that supports logical multiple concurrent communication pipes between end points.
- **Queue manager:** that supports applications through the provision of messaging and queuing capabilities.

Table 16 shows the relationship between these components and the connection style. The client-server connection style describes the situation where MQSeries Everyplace can operate in either client or server mode. The servlet option describes the case where MQSeries Everyplace is configured as an HTTP servlet with the HTTP server itself responsible for listening for incoming connection requests.

Table 16. Connection styles

	Queue manager	Channel manager	Channel listener
Client	Yes		
Client-server	Yes	Yes	Yes
Peer	Yes		
Server	Yes	Yes	Yes
Servlet	Yes	Yes	

MQSeries Everyplace applications are not directly aware of the connection style used by the queue managers. However style is significant in that it affects what resources are available to the parties, which queue managers can connect with other queue managers, the MQSeries Everyplace footprint, and which connections can concurrently exist.

Peer-to-peer connection

A peer-to-peer channel includes the capabilities of a channel manager and a channel listener for a single channel. When a peer-to-peer channel is created between two queue managers, one queue manager must act as a listener and the other as the connection initiator. A peer-to-peer connected queue manager can initiate multiple peer-to-peer connections with other queue managers, but it can only respond to one incoming connection request and then must wait for that peer-to-peer channel to be closed before responding to another such request. Over any one peer-to-peer channel the two participating queue

managers can both initiate actions, thus for example, applications on each queue manager can access queues on the other.

Peer-to-peer channels may not be usable through firewalls since the target of the incoming connection request may not be acceptable to the firewall.

Client-server connection

Standard channels, used for the client-server connection style, have no listening capabilities but depend on an independent listener at the server, and the server requires a channel manager to handle multiple concurrent channels. The client initiates the connection request and the server responds. A server can usually handle multiple incoming requests from clients. Over a standard channel the client has access to resources on the server. If an application on the server needs synchronous access to resources on the client, a second channel is required where the roles are reversed. However, since standard channels are themselves bidirectional, messages destined for a client from its server's transmission queue, are delivered to it over the standard (client-server) channel that it initiated.

A client can be a client to multiple servers simultaneously. (Note that a channel manager is not required to support this configuration because channel managers handle multiple inbound channels.)

The client-server connection style is generally suited for use through firewalls since the target of the incoming connection is normally identified as being acceptable to the firewall.

Multiple connection styles

A single queue manager can be capable of initiating either peer-to-peer or client-server connections, and of responding either as a server or a peer. In this case, the peer channel listener and the standard channel listener must have different port numbers.

Classes

MQSeries Everyplace provides a choice of classes for certain functions to allow the behavior of MQSeries Everyplace to be customized to meet specific application requirements. In some cases the interfaces to classes are documented so that additional alternatives can be developed. Table 17 on page 57 summarizes the possibilities. Classes can be identified either explicitly or through the use of alias names.

Table 17. Class options

Class	Alternates supplied	Interfaces documented
administration	no	yes
Authenticators	yes	no
Communications adapter	yes	yes
Communications style	yes	no
Compressors	yes	no
Cryptors	yes	no
Event log	sample provided	yes
Messages	no	yes
Queue storage	yes	no
Rules	default classes provided	yes
Trace	samples provided	yes

Application loading

When an MQSeries Everyplace queue manager is configured to operate as a client (or peer) the initiating application is responsible for loading any other applications into the JVM. Standard Java facilities can be used for this, or the class loader included as part of MQSeries Everyplace is available. Thus, multiple applications can run against a single queue manager in the same JVM. Alternatively multiple JVMs can be used but each requires its own queue manager and each of these must have a unique name.

When an MQSeries Everyplace queue manager is configured as a server MQSeries Everyplace is itself the initiating application. MQSeries Everyplace supports a preload class list and these classes are loaded in turn, before the queue manager is itself loaded.

Chapter 6. MQSeries Everyplace and MQSeries networks

Although an MQSeries Everyplace network can exist standalone, without the need for an MQSeries server or network, in practice MQSeries Everyplace is often used to complement an existing MQSeries installation, extending its reach to new platforms and devices, or providing advanced capabilities such as queue or message based security or synchronous messaging. From an MQSeries Everyplace application perspective, MQSeries queues and queue managers can be regarded as simply additional remote queues and queue managers. However, a number of functional restrictions exist because these queues are not accessed directly through MQSeries Everyplace dynamic channels and an MQSeries Everyplace queue manager, but require the involvement of an MQSeries Everyplace gateway. The gateway can send messages to multiple MQSeries queue managers either directly or indirectly, through MQSeries client channels. If the connection is indirect, the messages pass through MQSeries client channels to an intermediate MQSeries queue managers and then onwards through MQSeries message channels to the target queue manager.

Messages from an MQSeries application destined for MQSeries Everyplace are addressed to the MQSeries Everyplace queue manager and queue as normal, with the MQSeries routing (remote queue manager definitions) defined such that the MQSeries Everyplace messages arrive on specific MQSeries transmission queues. MQSeries channels are not defined for the transmissions queues, as would be normal practice, instead the MQSeries Everyplace gateway pulls the messages off these queues and ensures their delivery to the MQSeries Everyplace destination. The number of transmission queues to be used (that is the number of routes) is configurable and should be set to reflect the volume of messages to be delivered.

Interface to MQSeries

The architecture of MQSeries Everyplace supports the concept of one or more optional bridge types between MQSeries Everyplace and other messaging systems.

In MQSeries Everyplace Version 1.2 only one such bridge type is supported, the *MQSeries bridge* that interfaces between MQSeries Everyplace and MQSeries networks. This bridge uses the MQSeries Java client to interface to one or more MQSeries queue managers, thereby allowing messages to flow from MQSeries Everyplace to MQSeries and vice versa. In the current version of MQSeries Everyplace one such bridge is recommended per server, and each is associated with multiple *MQSeries queue manager proxies* (definitions of

MQSeries queue managers). A queue manager proxy definition is required for each MQSeries queue manager that communicates with MQSeries Everyplace. Each of these definitions can have one or more associated *client connection services*, where each represents a connection to a single MQSeries queue manager. Each of these may use a different MQSeries server connection to the queue manager, and optionally a different set of properties such as user exits or ports.

A gateway may have a number of *listeners* that use that gateway to connect to the MQSeries queue manager and retrieve messages from MQSeries to MQSeries Everyplace. A listener uses only one service to establish its connection, with each listener connecting to a single transmission queue on the MQSeries queue manager. Each listener moves messages from a single MQSeries transmission queue to anywhere on the MQSeries Everyplace network, via its parent gateway queue manager. Thus a single gateway queue manager can funnel multiple MQSeries message sources into the MQSeries Everyplace network.

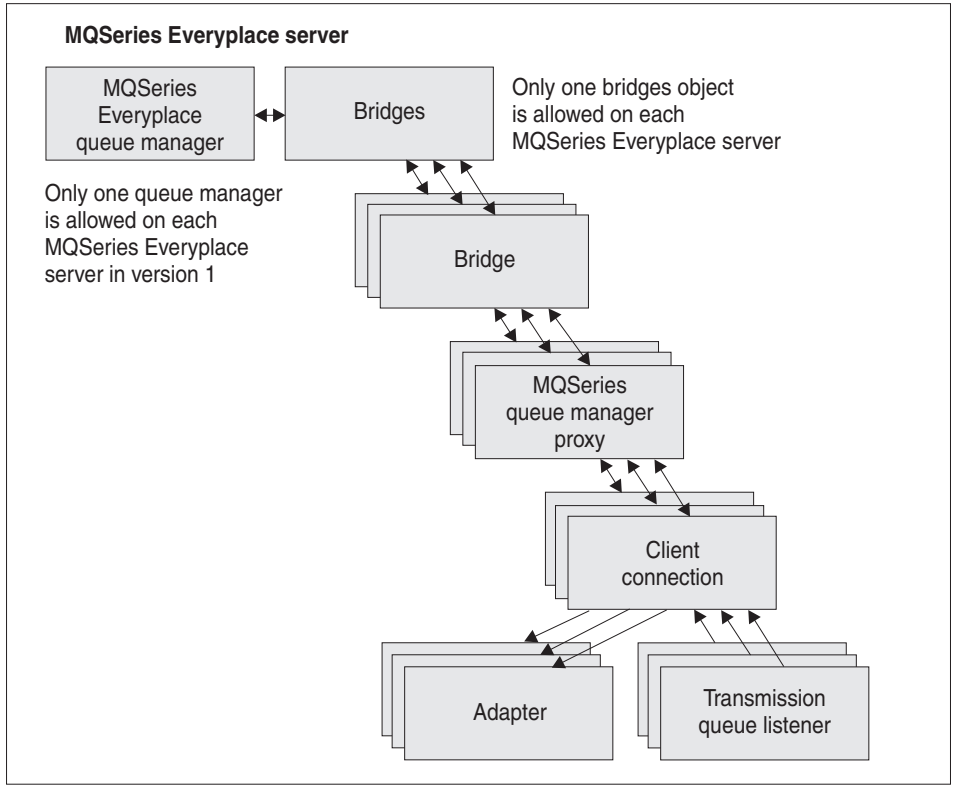


Figure 9. MQSeries Bridge object hierarchy

When moving messages in the other direction, from MQSeries Everyplace to MQSeries, the gateway queue manager configures one or more *bridge queue* objects. Each bridge queue object can connect to any queue manager directly and send its messages to the target queue. In this way a gateway can dispatch MQSeries Everyplace messages routed through a single MQSeries Everyplace queue manager to any MQSeries queue manager, either directly or indirectly. The bridges object has the properties shown in Table 18.

Table 18. Bridges object properties

Property	Explanation
Bridgename	List of bridge names
Run state	Status: running or stopped

The bridges object, and the other gateway objects can be started and stopped independently of the MQSeries Everyplace queue manager. If such a gateway object is started (or stopped) the action also applies to all of its children (all bridges, queue manager proxies, client connections, and transmission queue listeners). The bridge object has the properties shown in Table 19.

Table 19. Bridge properties

Property	Explanation
Class	Bridge class
Default transformer	The default class (rule class) to be used to transform a message from MQSeries Everyplace to MQSeries (or vice versa) if no other transformer class has been associated with the destination queue
Heartbeat interval	The basic timing unit to be used for performing actions against bridge objects
Name	Name of the bridge object
Run state	Status: running or stopped
Startup rule class	Rule class used when the bridge object is started
MQSeries Queue Manager Proxy Children	List of all Queue Manager Proxies that are owned by this bridge

In simple cases a default transformer (rule) can be used to handle all message conversions. Additionally a transformer can be set on a per listener basis (for messages from MQSeries to MQSeries Everyplace) that overrides this default. For more specific control the transformation rules can be set on a target queue basis using bridge queue definitions on the gateway. This applies both to MQSeries Everyplace and MQSeries target queues.

The MQSeries queue manager proxy holds the properties specific to a single MQSeries queue manager. The proxy properties are shown in Table 20.

Table 20. MQSeries queue manager proxy properties

Property	Explanation
Class	MQSeries queue manager proxy class
MQSeries host name	IP host name used to create connections to the MQSeries queue manager via the Java client classes. If not specified then the MQSeries queue manager is assumed to be on the same machine as the bridge and the Java bindings are used
MQSeries queue manager proxy name	The name of the MQSeries queue manager
Name of owning bridge	Name of the bridge object that owns this MQSeries queue manager proxy
Run state	Status: running or stopped
Startup rule class	Rule class used when the MQSeries queue manager object is started
Client Connection Children	List of all the Client Connection objects that are owned by this proxy

The gateway connection service definition holds the detailed information required to make a connection to an MQSeries queue manager. The connection properties are shown in Table 21.

Table 21. Client connection service properties

Property	Explanation
Adapter class	Class to be used as the gateway adapter
CCSID*	The integer MQSeries CCSID value to be used
Class	Bridge client connection service class
Max connection idle time	The maximum time a connection is allowed to be idle before being terminated
MQSeries password*	Password for use by the Java client
MQSeries port*	IP port number used to create connections to the MQSeries queue manager via the Java client classes. If not specified then the MQSeries queue manager is assumed to be on the same machine as the bridge and the Java bindings are used
MQSeries receive exit class*	Used to match the receive exit used at the other end of the client channel; the exit has an associated string to allow data to be passed to the exit code

Table 21. Client connection service properties (continued)

Property	Explanation
MQSeries security exit class*	Used to match the security exit used at the other end of the client channel; the exit has an associated string to allow data to be passed to the exit code
MQSeries send exit class*	Used to match the send exit used at the other end of the client channel; the exit has an associated string to allow data to be passed to the exit code
MQSeries user ID*	user ID for use by the Java client
Client connection service name	Name of the server connection channel on the MQSeries machine
Name of owning queue manager proxy	The name of the owning queue manager proxy
Startup rule class	Rule class used when the bridge client connection service object is started
Sync queue name	The name of the MQSeries queue that is used by the bridge for synchronization purposes
Sync queue purger rules class	The rules class to be used when a message is found on the sync queue
Run state	Status: running or stopped
Name of owning Bridge	The name of the Bridge object that owns this client connection
MQ XmitQ Listener Children	List of all the listener objects that use this client connection
*Details of these parameters can be found in the *MQSeries Using Java* documentation	

The *adapter class* is used to send messages from MQSeries Everyplace to MQSeries and the *sync queue* is used to keep track of the status of this process. Its contents are used in recovery situations to guarantee assured messaging; after a normal shutdown the queue is empty. It can be shared across multiple client connections and across multiple bridge definitions provided that the receive, send and security exits are the same. This queue can also be used to store state about messages moving from MQSeries to MQSeries Everyplace , depending upon the listener properties in use. The *sync queue purger rules class* is used when a message is found on the sync queue, indicating a failure of MQSeries Everyplace to confirm a message.

The maximum connection idle time is used to control the pool of Java client connections maintained by the bridge client connection service to its MQSeries system. When an MQSeries connection becomes idle, through lack of use, a timer is started and the idle connection is discarded if the timer expires before the connection is reused. Creation of MQSeries connections is an expensive

operation and this process ensures that they are efficiently reused without consuming excessive resources. A value of zero indicates that a connection pool should not be used.

The listener object, which moves messages from MQSeries to MQSeries Everyplace , has the properties shown in Table 22.

Table 22. Listener properties

Property	Explanation
Class	Listener class
Dead letter queue name	Queue used to hold messages from MQSeries to MQSeries Everyplace that cannot be delivered
Listener state store adapter	Class name of the adapter used to store state information
Listener name	Name of the MQSeries XMIT queue supplying messages
Owning client connection service name	Client connection service name
Run state	Status: running or stopped
Startup rule class	Rule class used when the listener object is started
Transformer class	Rule class used to determine the conversion of an MQSeries message to MQSeries Everyplace
Undelivered message rule class	Rule class used to determine action when messages from MQSeries to MQSeries Everyplace cannot be delivered
Seconds wait for message	An advanced option that can be used to control listener performance in exceptional circumstances

The *undelivered message rule class* determines what action is taken when a message from MQSeries to MQSeries Everyplace cannot be delivered. Typically it is placed in the *dead letter queue* of the MQSeries system.

In order to provide assured delivery of messages, the listener class uses the *listener state store adapter* to store state information, either on the MQSeries Everyplace system or in the sync queue of the MQSeries system.

The transmission queue listener allows MQSeries remote queues to refer to MQSeries Everyplace local queues. You can also create MQSeries Everyplace remote queues that refer to MQSeries local queues. These MQSeries Everyplace remote queue definitions are called *MQSeries-bridge queues* and they can be used to get, put and browse messages on an MQSeries queue.

An MQSeries-bridge queue definition can contain the following attributes.

Table 23. MQSeries-bridge queue properties

Property	Explanation
Alias names	Alternative names for the queue
Authenticator	Must be null
Class	Object class
Client connection	Name of the client connection service to be used
Compressor	Must be null
Cryptor	Must be null
Expiry	Passed to transformer
Maximum message size	Passed to the rules class
Mode	Must be synchronous
MQ queue manager proxy	Name of the MQSeries queue manager to which the message should first be sent
MQSeries bridge	Name of the bridge to convey the message to MQSeries
Name	Name by which the remote MQSeries queue is known to MQSeries Everyplace
Owning queue manager	Queue manager owning the definition
Priority	Priority to be used for messages (unless overridden by a message value)
Remote MQSeries queue name	Name of the remote MQSeries queue
Rule	Rule class used for queue operations
Queue manager target	MQSeries queue manager owning the queue
Transformer	Name of the transformer class that converts the message from MQSeries Everyplace format to MQSeries format
Type	MQSeries bridge queue

Note: The *cryptor*, *authenticator*, and *compressor* classes define a set of queue attributes that dictate the level of security for any message passed to this queue. From the time on MQSeries Everyplace that the message is sent initially, to the time when the message is passed to the MQSeries-bridge queue, the message is protected with at least the queue level of security. These security levels are *not* applicable when the MQSeries-bridge queue passes the message to the MQSeries system, the security send and receive exits on the client connection are used during this transfer. No checks are made to make sure that the queue level of security is maintained.

MQSeries-bridge queues are synchronous only. Asynchronous applications must therefore use either a combination of MQSeries Everyplace store-and-forward and home-server queues, or asynchronous remote queue definitions as an intermediate step when sending messages to MQSeries-bridge queues.

Applications make use of MQSeries-bridge queues like any other MQSeries Everyplace remote queue, using the `putMessage`, *browseMessages* and *getMessage* methods of the MQeQueueManager class. The queue name parameter in these calls is the name of the MQSeries-bridge queue, and the queue manager name parameter is the name of the MQSeries queue manager. However, in order for this queue manager name to be accepted by the local MQSeries Everyplace server, a connection definition with this MQSeries queue manager name must exist with null for all the parameters, including the channel name.

Note: there are some restrictions on the use of getMessage and browseMessages with MQSeries-bridge queues. It is not possible to get or browse messages from MQSeries-bridge queues that point to MQSeries remote queue definitions. Nor is it possible to use nonzero Confirm IDs on MQSeries-bridge queue gets. This means that the getMessage operation on MQSeries-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 MQSeries.

Administration of the MQSeries-bridge is handled in the same way as the administration of a normal MQSeries Everyplace queue manager - through the use of administration messages. New classes of messages are defined as appropriate to the managed object. Table 13 on page 39 shows the gateway administration message classes.

Message conversion

MQSeries Everyplace messages destined for MQSeries pass through the bridge and are converted into an MQSeries format, using either a default transformer or one specific to the target queue. A custom transformer offers much flexibility, for example it would be good practice to use a subclass of the MQSeries Everyplace message object class to represent messages of a particular type over the MQSeries Everyplace network. On the gateway a transformer could convert the message into an MQSeries format using whatever mapping between fields and MQSeries values that was appropriate as well as add specific data to represent the significance of the subclass.

The default transformer from MQSeries Everyplace to MQSeries cannot take advantage of subclass information but has been designed to be useful in a wide range of situations. It has the following characteristics:

- **Message flow from MQSeries Everyplace to MQSeries:**

  The default transformer from MQSeries Everyplace to MQSeries works in conjunction with the MQeMQMsgObject class. This class is a representation of all the fields you could find in an MQSeries message header. Using the MQeMQMsgObject, your application can set values (priority for example) using set() methods. Thus, when an MQeMQMsgObject (or an object derived from the MQeMQMsgObject class) is passed through the default MQSeries Everyplace transformer, the default transformer (MQeBaseTransformer) gets the values from inside the MQeMSMsgObject, and sets the corresponding values in the MQSeries message (for example, the priority value is copied over to the MQSeries message).

  If the message being passed is not an MQeMQMsgObject, and is not derived from the MQeMQMsgObject class, the whole MQSeries Everyplace message is copied into the body of the MQSeries message (*funneled*). The message format field in the MQSeries message header is set to indicate that the MQSeries message holds a message in MQSeries Everyplace ″funneled″ format.

- **MQSeries to MQSeries Everyplace message flow:**

  MQSeries messages for MQSeries Everyplace are handled similarly to those travelling in the other direction. The default transformer inspects the message type field of the MQSeries header and acts accordingly.

  If the MQSeries header indicates a ″funneled″ MQSeries Everyplace message, then the MQSeries message body is reconstituted as the original MQSeries Everyplace message that is then posted to the MQSeries Everyplace network.

  If the message is not a ″funneled″ MQSeries Everyplace message, then the MQSeries message header content is extracted, and placed into an MQeMQMsgObject object. The MQSeries message body is treated as a simple byte field, and is also placed into the MQeMQMsgObject object. The MQeMQMsgObject is then posted to the MQSeries Everyplace network.

This MQeMQMsgObject class and the default transformer behavior mean that :

- An MQSeries Everyplace message can travel across an MQSeries network to an MQSeries Everyplace network without change.
- An MQSeries message can travel across an MQSeries Everyplace network to an MQSeries network without change.
- An MQSeries Everyplace application can drive any existing MQSeries application without the MQSeries application being changed.

Function

MQSeries remote queues are enabled for synchronous MQSeries Everyplace put messaging operations, from an MQSeries Everyplace queue manager; all other messaging operations must be asynchronous.

MQSeries Everyplace administration messages cannot be sent to an MQSeries queue manager. The AdminQ does not exist there and the administration message format differs from that used by MQSeries.

Compatibility

An MQSeries Everyplace network can exist independently of MQSeries, but in many situations the two products together are needed to meet the application requirements. MQSeries Everyplace can integrate into an existing MQSeries network with compatibility including the aspects summarized below:

Addressing and naming:

- identical addressing semantics using a queue manager/queue address
- Common use of an ASCII name space

Applications:

MQSeries Everyplace is able to support existing MQSeries applications without application change.

Channels:

MQSeries Everyplace gateways use MQSeries client channels.

Message interchange and content:

- interchange of messages between MQSeries Everyplace and MQSeries
- message network invisibility (messages from either MQSeries Everyplace or MQSeries can cross the other network without change)
- mutual support for identified fields in the MQSeries message header
- once-only assured message delivery

MQSeries Everyplace Version 1.2 does not support all the functions of MQSeries. Apart from environmental, operating system and communication considerations, some of the more significant differences are detailed below. Note however that within MQSeries Everyplace many application tasks can be achieved through alternative means using MQSeries Everyplace features, or through the exploitation of sub-classing, the replacement of the supplied classes or the exploitation of the rules, interfaces and other customization features built into the product.

- No clustering support
- No distribution list support
- No grouped/segmented messages
- No load balancing/warm standby capabilities
- No reference message
- No report options
- No shared queue support
- No triggering
- No unit of work support, no XA-coordination

Scalability and performance characteristics are different.

Assured delivery

Although both MQSeries Everyplace and MQSeries offer assured delivery, they each provide for different levels of assurance. When a message is travelling from MQSeries Everyplace to MQSeries, the message transfer is only assured if the combination of putMessage and confirmPutMessage is used (see "Queue manager configuration" on page 33). When a message is travelling from MQSeries to MQSeries Everyplace, the transfer is assured only if the MQSeries message is defined as persistent.

Chapter 7. Programming interfaces

The *MQSeries Everyplace Systems Programming Interface (SPI)* is the programming interface to MQSeries Everyplace . Two languages are supported, Java and C:

The Java version provides access to all MQSeries Everyplace functions. The detailed classes, methods and procedures are detailed in the *MQSeries Everyplace for Multiplatforms Programming Reference*; examples of MQSeries Everyplace programming are given in the *MQSeries Everyplace for Multiplatforms Programming Guide*.

The C support for Palm provides access for a subset of the MQSeries Everyplace function for use on Palm devices. Details of these classes and procedures, together with programming guidance is provided in *MQSeries Everyplace Native Client Information*

Chapter 8. Getting started with MQSeries Everyplace

MQSeries Everyplace is a family of products that collectively provide the tools needed to develop, deploy and manage MQSeries Everyplace messaging and queuing solutions. The family compromises:

1. The *MQSeries Everyplace licensed product* (available on physical media from IBM or as a Web download from *http://software.boulder.ibm.com/dl/mqsem/mqsem-p*). The licensed product includes:
   - MQSeries Everyplace Java classes
   - Helper classes
   - Application source code examples
   - Utilities
   - Reference manuals
   - License information

   The physical Program Product also includes entitlement to use the product for non-development (productive) use on certain platforms. Further capacity units need to be purchased for use on larger machines, or with the MQSeries-bridge.
2. MQSeries Everyplace SupportPacs (available as Web downloads from *http://software.boulder.ibm.com/dl/mqsem/mqsem-p* (as above) or from *http://www.ibm.com/software/mqseries/everyplace*). These are essential supplements to the licensed product and include for example:

   EAP1: MQSeries Everyplace - Device code for the Palm OS
   C programming language support for MQSeries Everyplace Version 1.0.1 application development on the Palm OS. (This code is also included on the product CD inside the file eap1.zip)

   EP01: MQSeries Everyplace - Performance Report
   Analyses MQSeries Everyplace performance on a variety of client platforms

   ES01: MQSeries Everyplace - Administration Tool (MQeExplorer v1.0)
   A generic tool for all Java platforms enabling easy graphical administration of MQSeries Everyplace queue managers

   ES02: MQSeries Everyplace - Explorer (MQe_Explorer v1.2)
   An MQSeries Everyplace administration tool developed exclusively to support the Microsoft Windows range of operating systems

The management tools in the MQSeries Everyplace SupportPacs play an important role in all phases of application development and rollout. They are more sophisticated than the utilities included with the licensed product and are an essential aid to getting started, configuring , inspecting pilot networks, and in managing production systems.

Using MQSeries Everyplace

Given the wide range of uses for MQSeries Everyplace, the product is not installed, configured and deployed in the same way as other members of the MQSeries family. The underlying concept here is that typically there are three phases in the adoption of MQSeries Everyplace by an enterprise:

1. **Development and prototyping phase**

   In the early learning, development, and prototyping phase, the MQSeries Everyplace product is available for installation and use without charge - subject to the conditions of the IBM MQSeries Everyplace development license. MQSeries Everyplace applications are developed, using the functions of the MQSeries Everyplace Java classes and C routines. These applications can be packaged in a variety of ways, for example:

   - An MQSeries Everyplace queue manager can be set up as a daemon with one or more applications launched into the same Java virtual machine and sharing a common queue manager.
   - The application embeds the required MQSeries Everyplace classes such that the application runs on machines where MQSeries Everyplace has not been installed, launching its own queue manager into its own JVM.
   - The application uses the MQSeries Everyplace classes that exist on the target machine.

   Support from IBM is not included with the development license. However, support during application development and beyond is provided with the deployment license (see below).

2. **Deployment phase**

   The deployment phase represents the rollout and use of the developed applications and therefore, under the terms of the IBM MQSeries Everyplace license, capacity units are required to use the product. The classes may only be distributed with the application with agreement from IBM, or where the users already have entitlement to use them. Otherwise users must provide the necessary classes themselves.

3. **Management phase**

   Subsequently, when MQSeries Everyplace queue managers are active within a network, tools are needed to inspect and manage them.

Support for MQSeries Everyplace is provided under the terms of the International Program License Agreement.

This adoption life cycle is the justification for the variation in level of support with platforms. For the MQSeries Everyplace with capacity units (and Category 3 SupportPacs) IBM distinguishes between:

- Platforms where the installation and application development is supported:
  - problem reports on install and/or application development and use will be accepted
- Platforms where the application deployment is permitted but not directly supported:
  - problem reports may be required to be reproduced on a supported platform
- Platforms where application deployment is supported:
  - problem reports resulting from application deployment will be accepted

Gaining experience

There are many ways to get started with MQSeries Everyplace. Experience suggests that getting a queue manager up and running, followed by a simple MQSeries Everyplace network, is a productive way to become familiar with the product and its concepts. Then writing a simple application is sound preparation for in depth study of the product details. In the early stages it is generally not helpful to examine other members of the MQSeries family. Later, when the bridge functionality is of interest, this understanding becomes essential.

With this strategy in mind, new users are recommended to adopt the following approach:

Understand the essentials of the concepts presented in *MQSeries Everyplace for Multiplatforms Introduction* (this book) and then do the following :

1. If you have access to a machine running a Microsoft Windows operating system, download the MQe_Explorer, SupportPac ES02 (MQe_Explorer v 1.2). You do not have to install the licensed product beforehand, but if you do not, you are restricted to development use by the terms of the license.
2. Follow the specific instructions given below in "First use of the ES02: MQe_Explorer".

First use of the ES02: MQe_Explorer

The MQe_Explorer comes with a comprehensive User Guide that describes fast ways of getting a first queue manager configured. The manual is generally recommended to a wider audience than just MQe_Explorer administrators because it includes a number of sample scripts that illustrate

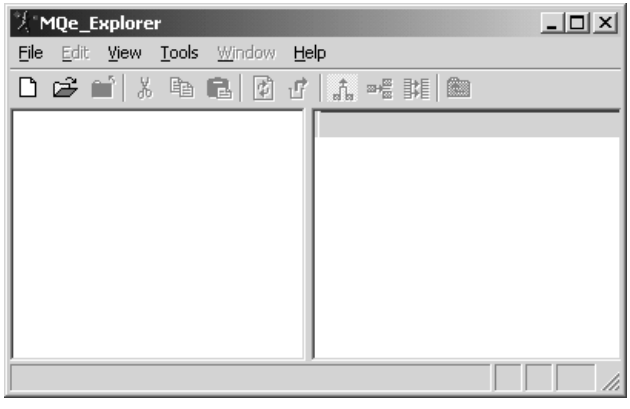
important examples of MQSeries Everyplace operations. The SupportPac includes two executable versions of MQe_Explorer:

MQe_ExplorerX.exe
This version embeds within the .exe file all the MQSeries Everyplace classes that are needed for its operation. MQe_ExplorerX.exe runs without MQSeries Everyplace having been installed on the machine. It is ideal for first time users.

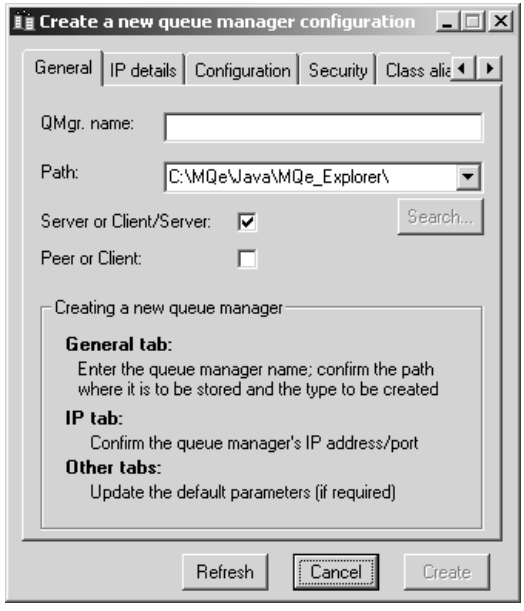
MQe_Explorer.exe
This version depends upon MQSeries Everyplace having been previously installed. The advantages are that it picks up the latest level of the MQSeries Everyplace libraries and is much smaller. It is intended for developers and administrators.

As an example of the ease with which queue managers can be created, the following abridged instructions show the power of MQSeries Everyplace.

1. Double click the MQe_ExplorerX.exe icon .
   A message indicates that no saved options have been found, click **OK** (this message will not occur again). The following administrative window is displayed:

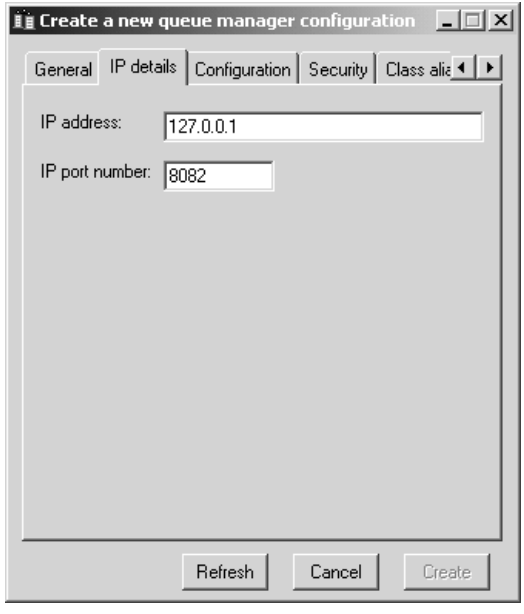


2. Click the new icon on the toolbar.
   This creates a new queue manager. The following window is displayed:



3.

a. Type in a queue manager name (e.g. `SampleQM`).
b. Select the **IP details** tab

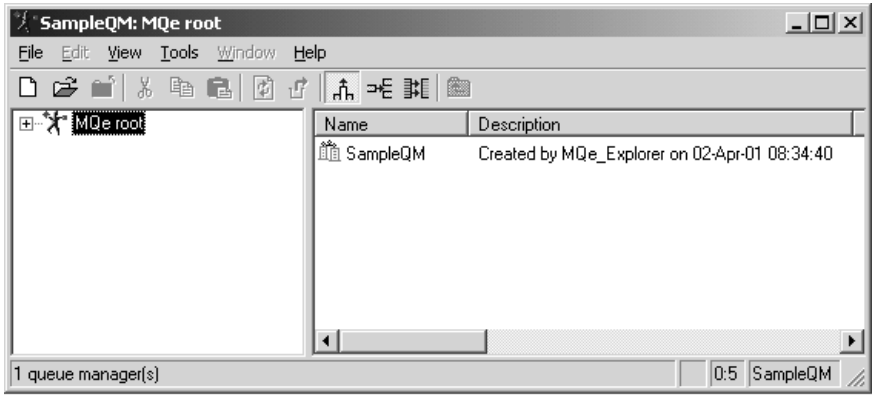


4.

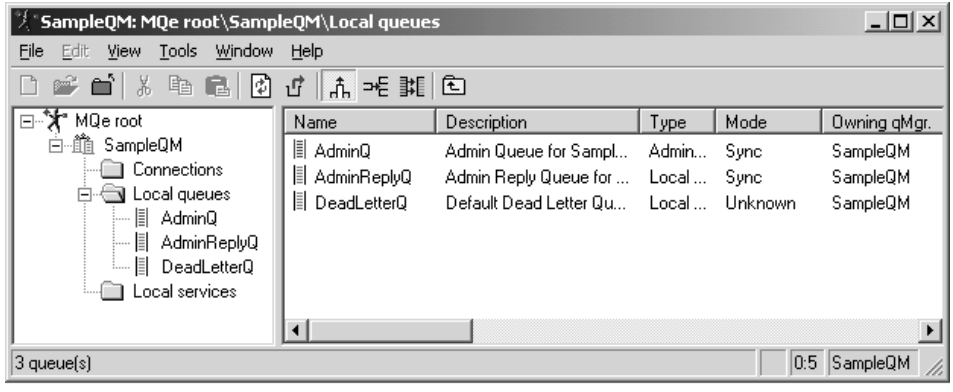
a. Type in an IP address - the address is the IP address of the machine being used, but a good value to type in at this stage is `127.0.0.1` (local host).

b. Click the **Create** button. A message is issued identifying an initialization file that has been created (the name is needed for future access to this queue manager);

c. Click **OK** and the following window is displayed:



5. A server queue manager is created, executing in its own JVM. It is listening on port 8081 for incoming client/server channel connection requests. If the + symbols are all expanded in the tree in the left hand pane and the window and panes are resized, it can be seen that four queues are created.



6. To experiment more, follow the instructions in the *MQe_Explorer User Guide*. You can create queues, connections, messages and even complete

MQSeries Everyplace networks. Applications can be loaded into this same queue manager and can be run concurrently with MQe_Explorer.

Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire
England
SO21 2JN

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Trademarks

The following terms are trademarks of International Business machines Corporation in the United States, or other countries, or both.

AIX AS/400 IBM MQSeries OS/390

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

This glossary describes terms used in this book and words used with other than their everyday meaning. In some cases, a definition may not be the only one applicable to a term, but it gives the particular sense in which the word is used in this book.

If you do not find the term you are looking for, see the index or the *IBM Dictionary of Computing*, New York:. McGraw-Hill, 1994.

Application Programming Interface (API). An Application Programming Interface consists of the functions and variables that programmers are allowed to use in their applications.

asynchronous messaging. A method of communicating between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with *synchronous messaging*.

authenticator. A program that checks that verifies the senders and receivers of messages.

bridge. An MQSeries Everyplace object that allows messages to flow between MQSeries Everyplace and other messaging systems, including MQSeries.

channel. See *dynamic channel* and *MQI channell*.

channel manager. An MQSeries Everyplace object that supports logical multiple concurrent communication pipes between end points.

class. A class is an encapsulated collection of data and methods to operate on the data. A class may be instantiated to produce an object that is an instance of the class.

client. In MQSeries, a client is a run-time component that provides access to queuing services on a server for local user applications.

compressor. A program that compacts a message to reduce the volume of data to be transmitted.

cryptor. A program that encrypts a message to provide security during transmission.

dynamic channel. A dynamic channel connects MQSeries Everyplace devices and transfers synchronous and asynchronous messages and responses in a bidirectional manner.

encapsulation. Encapsulation is an object-oriented programming technique that makes an object's data private or protected and allows programmers to access and manipulate the data only through method calls.

gateway. An MQSeries Everyplace gateway (or server) is a computer running the MQSeries Everyplace code including a channel manager.

Hypertext Markup Language (HTML). A language used to define information that is to be displayed on the World Wide Web.

instance. An instance is an object. When a class is instantiated to produce an object, we say that the object is an instance of the class.

interface. An interface is a class that contains only abstract methods and no instance variables. An interface provides a common set of methods that can be implemented by subclasses of a number of different classes.

Internet. The Internet is a cooperative public network of shared information. Physically, the Internet uses a subset of the total resources of all the currently existing public telecommunication networks. Technically, what distinguishes the Internet as a cooperative public network is its

use of a set of protocols called TCP/IP (Transport Control Protocol/Internet Protocol).

Java Developers Kit (JDK). A package of software distributed by Sun Microsystems for Java developers. It includes the Java interpreter, Java classes and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

Java Naming and Directory Service (JNDI). An API specified in the Java programming language. It provides naming and directory functions to applications written in the Java programming language.

Lightweight Directory Access Protocol (LDAP). LDAP is a client-server protocol for accessing a directory service.

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 an MQSeries client to a queue manager on a server system and transfers MQI calls and responses in a bidirectional manner.

MQSeries. MQSeries is a family of IBM licensed programs that provide message queuing services.

object. (1) In Java, an object is an instance of a class. A class models a group of things; an object models a particular member of that group. (2) In MQSeries, an object is a queue manager, a queue, or a channel.

package. A package in Java is a way of giving a piece of Java code access to a specific set of classes. Java code that is part of a particular package has access to all the classes in the package and to all non-private methods and fields in the classes.

personal digital addistant (PDA). A pocket sized personal computer.

private. A private field is not visible outside its own class.

protected. A protected field is visible only within its own class, within a subclass, or within packages of which the class is a part

public. A public class or interface is visible everywhere. A public method or variable is visible everywhere that its class is visible

queue. A queue is an MQSeries object. Message queueing applications can put messages on, and get messages from, a queue

queue manager. A queue manager is a system program the provides message queuing services to applications.

server. (1) An MQSeries Everyplace server is a device that has an MQSeries Everyplace channel manager configured. (2) An MQSeries server is a queue manager that provides message queuing services to client applications running on a remote workstation. (3) More generally, a server is a program that responds to requests for information in the particular two-program information flow model of client/server. (3) The computer on which a server program runs.

servlet. A Java program which is designed to run only on a web server.

subclass. A subclass is a class that extends another. The subclass inherits the public and protected methods and variables of its superclass.

superclass. A superclass is a class that is extended by some other class. The superclass's public and protected methods and variables are available to the subclass.

synchronous messaging. A method of communicating between programs in which

programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing . Contrast with *asynchronous messaging*.

Transmission Control Protocol/Internet Protocol (TCP/IP). A set of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

Web. See World Wide Web.

Web browser. A program that formats and displays information that is distributed on the World Wide Web.

World Wide Web (Web). The World Wide Web is an Internet service, based on a common set of protocols, which allows a particularly configured server computer to distribute documents across the Internet in a standard way.

Bibliography

Related publications:

- *MQSeries Everyplace for Multiplatforms Read Me First*, GC34–5862
- *MQSeries Everyplace for Multiplatforms Programming Reference*, SC34–5846
- *MQSeries Everyplace for Multiplatforms Programming Guide*, SC34–5845
- *MQSeries Everyplace for Multiplatforms Native Client Information*, GC34–5883
- *MQSeries An Introduction to Messaging and Queuing*, GC33-0805-01
- *MQSeries for Windows NT V5R1 Quick Beginnings*, GC34-5389-00

Index

Q

R

S

T

W

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM®.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To 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 the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

GC34-5843-02